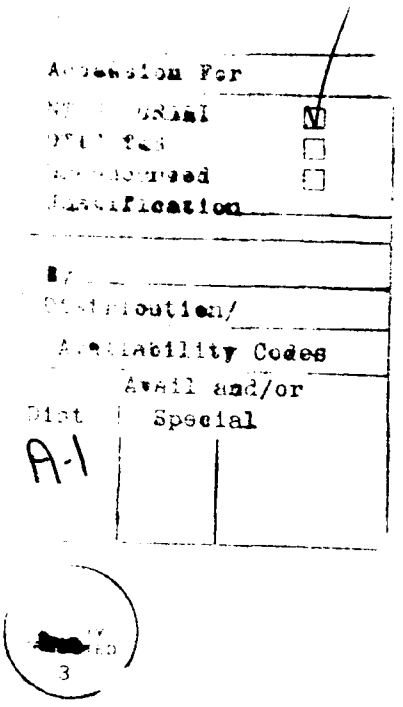
The Rhet Programmer's Guide (for Rhet Version 17.9)

Bradford W. Miller

Technical Report 363
December 1990

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE



The Rhet Programmer's Guide (For Rhet Version 17.9)

Bradford W. Miller

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 363 (Originally TR239)

December 1990

This work was supported in part by ONR research contract no. N00014-80-C-0197, in part by U.S. Army Engineering Topographic Laboratories research contract no. DACA76-85-C-0001, and in part by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under Contract No. F30602-85-C-0008.

Abstract

Rhetorical (Rhet) is a programming / knowledge representation system that offers a set of tools for building an automated reasoning system. It's emphasis is on flexibility of representation.

This document extends [Miller, 1990] with more information about the internals of the Rhet system. In addition it provides the information needed for users to write their own builtin functions, or better lispfns (that use internally provided library functions).

Contents

Chapter 1

Introduction

1.1 What is this thing?

The intention of this document is to give an overview of the code of the Rhetorical (*Rhet*) system, and a tutorial overview of how the user may write builtins and lispfns. It is intended to be used by those who will maintain or extend the system. It is not intended for casual users of the system, nor is it an effective substitute for reading the code itself. This document is current for version 17.9 of Rhet, and should be updated periodically to reflect Rhet's current state.

The philosophy of this document is: if you want to know what the code does, read the code. This document is mainly intended to give a broad overview of the entire system's design to make reading the code possible. While most interface-level functions a builtin or lispfn author should need will be described herein, most internal functions are not. The maintainer is advised to use a machine that will allow him to get to function definitions or present documentation strings, as this will greatly ease program understanding.

1.2 Overview

To begin, then, a quick overview:

Rhet was originally intended to be primarily a rewrite of the *HORNE* [Allen and Miller, 1986] reasoning system. The intent of the rewrite was to:

- Provide an extended version of HORNE including extensions that had been planned to HORNE to support the current (local) research. This involves:
  1. Enhancement of the existing HORNE system to provide for contexts.
  2. Enhancement to provide for reasoning about negation (rather than simply negation by failure).
  3. Enhancement to provide some support for default reasoning[1].

[1] This goal has yet to be met.

4. Enhancement to provide some type of TMS.
5. Enhancement to provide a better user interface - both one that allows the user to debug Rhet code more easily, but also a more interactive form of the interface in keeping with Lisp Machine philosophy. (*E.g.* usage of ZMACS rather than a separate form editor).
6. Enhancement to support user defined *specialized* reasoners between specific types. For example, the TEMPOS system would be an example of a specialized reasoner.

- Provide a stable base for future enhancements to the reasoning system.
- Provide a more efficient implementation of the existing system, building on our experience with the old system, and specializing the new system for the Common Lisp language primarily, and the Lisp Machines secondarily. Version 17.0 still has some lisp machine dependant code, however with the availability of CLIM, CLOS and Pitman's condition system in 1991 on SUN based environments, we expect the next version to be ported to one of these environments (specifically Allegro Common Lisp with CLIM).

Note that since the original rewrite, Rhet's charter has been expanded to include better support for structured types. type calculus, objects of a distinguished subtype, inequality, *etc.*.

1.3 System Compartmentalization

To support maintenance and extensibility, the Rhet system was designed as a compartmentalized system spanned several packages (in the common-Lisp sense). In theory, the control of imports and exports into packages would help define the interface between packages.

Note that the ZetaLisp SCT:Defsystem facility is taken advantage of, as are SCT:Initializations. More specifically, CL-User:*Rhet-Initializations* is a list of forms to be evaluated (once) after loading Rhet. These are declared in the source via. SI:Add-Initialization forms. See the "Porting" Appendix. It is likely that in the portable implementation (v18?) this will be taken care of using a different mechanism.

1.3.1 Assert

The *Assertion Interface* provides fundamentally a user interface for adding axioms, equalities, or other forms of knowledge to Rhet. It is intended to be used either programmatically, or directly by the user via a enhanced interface.

1.3.2 E-Unify

The *E-Unification Subsystem* implements the actual unification engine for Rhet. It supports typed rvariable and equalities for detecting if two horn clauses unify.

1.3.3 Query

The *Query Interface* provides (separately) both a programmatic and user-enhanced interface for requesting proofs or doing unifications from Rhet.

1.3.4 RAX

The *Rhet Axiom Subsystem* is the keeper of forward and backward chaining axioms as defined by the user. It is responsible for adding and deleting axioms, as well as retrieving them as specified by the Reasoner or user.

1.3.5 Reasoner

The *Reasoner* subsystem implements the actual prover for Rhet. It accepts requests to prove a Horn Clause, and invokes the Unifier as needed, selecting axioms from the Backward Chaining Axiom Database, and executing Forward Chaining whenever new facts have been asserted. This is the actual 'Proof Engine' of the system.

1.3.6 Rhet-Terms

This package contains code that is intended to be the basic KB for the Rhet system, as well as utility functions used throughout Rhet that interface to a limited extent with the user (e.g. it provides the printers for the various Rhet objects, the definitions and handlers for rhet facts, terms, and types). It is also the keeper of equalities and assertions relative to contexts. Additionally, it supports adding and deleting assertions, adding and deleting contexts, and adding equalities[2].

1.3.7 RWin

The *Window Interface* contains the functions associated with the high-level user interface, that are not contained in some other package (*e.g.* on the Symbolics, the rhet editing mode is in the Zwei package). We are planning to have this system optionally loadable; Rhet supporting a tty-only interface on pure common lisp systems such as AKCL, where no windowing substrate is provided.

1.3.8 TypeA

The *Type Assertion Interface* takes type generation requests from the user, and installs them into the type database accessed by routines in Rhet-Terms.

[2]Equalities cannot be deleted.

1.3.9 UI

The *User Interface Common High Level* contains utility functions shared by the various user interface packages. It includes the parser, for example.

Chapter 2

Tutorial — Lisp Predicates

2.1 Where to use a Lispfn

Lispfns, that is lisp functions that are to be called from Rhet and act as predicates or assertions, allow for a very simple and convenient way to interface Rhet to some lisp system. This system may be:

- a primitive backend, acting as an efficient representation mechanism for a particular kind of knowledge that Rhet will want to manipulate, *e.g.* time intervals, parsed sentences, ...:
- a way to imporve performance when a predicate can fold naturally and easily into Lisp code that does not require the power of a builtin.

Lispfns, however, are much more limited than builtins. Their advantage is that they are much easier to write and debug than builtins, and require no deep understanding of the Rhet system as builtins do.

These limitations are:

- Lispfns cannot backtrack, builtins can. That is, a lispfn may only succeed or fail, and will be considered deterministic. They cannot bind any variable[1], because that can only be done in the context of backtrack handling: one provides code that handles the backtrack and tacks it onto the variable so should the variable need to be unbound this code can be called. Since lispfns do not provide this function, they cannot bind a variable.

[1] Partially because this would potentially lead to backtracking, but really because the variable binding mechanisms associate with bound variables certain information that is not provided by the simple lispfn mechanism and requires the much more complex macros used by the builtins. It is, in fact, relatively straightforward to write a builtin with the limited functionality of a lispfn, but that can bind variables, however since this involves a more careful understanding of the stack and how Rhet passes variables and does proofs lispfns have been limited to not take variables. Invoking a lispfn with a variable unbound will invoke the debugger. This should not be considered a serious limitation since the builtins `[GenValue]` and `[SetValue]` do have the capability of backtracking and binding a variable.

- Lispfns cannot have side effects. When Rhet does a proof, it may call a lispfn one or more times in the process of doing a proof. There is no guarentee of which or any of these calls will end up being logically responsible for the final proof(s) returned to the user. Thus if the lispfn has side effects it may do things that are unrelated to the actual thing proven, they will only be related to the means by which they were proven. For example:

(2.1)
```
[[P ?x] < [Q ?x ?y]
          [My-Lispfn ?y]
          [R ?y]]
```

It is possible in 2.1 to have My-Lispfn invoked on each value of ?y for which `[Q ?x ?y]` is true, though only one value of ?y allows `[R ?y]` to be provable. This last one is the only effective one used to prove `[P ?x]`, but it is not distinguishable by My-Lispfn from the other proofs. They may have been proofs from different parts of the proof tree, or multiple-proofs due to a Query:Prove-All. Further, the rule that tried to prove (directly or indirectly) `[P ?x]` may fail, and no notification is given to functions called from parts of the proof tree that get lopped off.

2.2 Writing Lispfns

Write a lispfn just like you were writing a lisp predicate. The easiest thing to use would be the Assert:DefRhetPred form, which is documented in [Miller, 1990]. Since you are making the assumption that all arguments to the lispfn will be bound when you are called, most predicates are quite straightforward. For example, suppose we wish to add date-string recognition to Rhet. Rhet already knows about strings to a limited extent, so we might just do:

(2.2)
```
(DefRhetPred Date-String? (input-string)
    "Takes a string and returns non-nil if it is a date"
    (time:parse input-string))
```

This works to the extent that input-string always represents a legal time, since Time:Parse will drop into the debugger otherwise. I leave it as an exercise to the reader to write a version of Date-String? that would actually return NIL if the input string were not a legal time.

2.3 Language Constructs

These definitions also appears in the User's Manual:

Rllib:Declare-Lispfn *<Name> Query-Function-Symbol* &Optional *Assert-Function-Symbol Type-Declarations*

Declares to Rhet that Name is not a predicate but a Lisp function; Rhet will reccgnize those <Name>s as calls to Lisp functions. If the Reasoner is attempting to prove an axiom that has been declared a Lisp function, it will call the Query-Function-Symbol (passed as a symbol to allow it to be incrementally recompiled). If it attempting to add a predicate to the KB whose head is declared with an associated Assert-Function, it will call the Assert-Function-Symbol rather than add it[2]. Lisp Query-Functions should only return "t" or "nil" which will be interpreted as true and false respectively. The optional Type-Declarations are a list of symbols representing the types of the arguments expected by the lispfn. This will be used for runtime debugging interaction, and as a hint to the compiler. Declare-Lispfn will check in the KB to see if Name has previously been used in constructing Rhet predicates, and if so, warn the user about potential problems[3].

Assert:DefRhetPred *Predicate-Name* (Argument-Lambda-List) &Body *Body*

Defines a Rhet predicate Predicate-Name that takes arguments according to the Argument-Lambda-List. Body is either a series of indicies and RHS definitions, or lisp code. These may not be mixed. Only the lispfn usage of DefRhetPred is described here.

The Argument-Lambda-List may be made up of the following:

1. & keywords, specifically:

   &Any the following variables (until the next & keyword) may be bound or unbound when the predicate is invoked, or may be bound to terms that are not fully ground. This is the default when no & keyword appears.

   &Bound the following variables are guarenteed by the programmer to be bound when the predicate is invoked. It is an error to attempt to prove the predicate without passing a fully ground term in this position. If (Declare (Optimize Safety)) appears, erroneous usage will signal an error[4]. If forms follow this keyword, variables embedded in the forms must be bound.

   &Forward [5] this must be the last &Keyword specified. If present, the DefRhetPred defines FC axiom(s) rather than BC. The form(s) following this key are the trigger(s). If none are present, it is the same as having specified :All as the trigger. No unbound variables may appear in the trigger. The other part of the lambda list becomes a pattern for the fact to be asserted if the body of the DefRhetPred is proved[6].

---

[2]It will still forward chain as if it had added it, although it may not detect a loop!

[3]A form that has been read by the parser before this declaration will have a head that is a Fact structure, and after will have a head that is the Name symbol. This distinction is picked up by the prover, so the former will never be recognized as a Lispfn at proof time, and the latter never looked up in the KB.

[4]Future functionality

[5]Currently unsupported with lispfns.

[6]Currently, it is an error to have both &forward, and the body containing a lisp function.

&Unbound the following variables are guarenteed by the programmer to be unbound when the predicate is invoked. If forms follow the keyword, variables within the forms must be unbound. It is an error to attempt to prove the predicate with any of the variables bound. If (Declare (Optimize Safety)) appears, erroneous usage will signal an error[7].

&Rest the following variable must be of type T-List (it will be changed if it is not), and will be bound to all the remaining arguments of the predicate when attempting to prove it. This is just like the normal usage of &Rest in forms.

2. Rhet variables, which have the properties of the closest preceeding & keyword, as above.
3. Rhet forms, which are pattern matched against the form being proved to see if this predicate is applicable.

The body then consists of the following:

1. If the first object is a string, it is considered a documentation string for the predicate.
2. If the first object (after the optional documentation string, if any) is a CL:Declare form, it is taken to be declarations about the predicate, as for CL. All CL declarations are legal for lispfn predicates. Other declarations that are legal include Foldable, Non-Foldable, Monotonic, and Non-Monotonic.
3. After the optional declarations, if the next object is not an index, the remainder to the body is taken to be an implicit Progn that defines an anonymous lisp function that will act as the predicate. At run-time, the lisp function will be evaluated, and if it returns non-nil, the predicate succeeds. Note that only one Assert:DefRhetPred may be defined on a predicate if it is to expand into a lisp function. Note that if the lisp function will have embedded Rhet forms that contain references to the variables in the Argument-Lambda-List, the DefRhetPred must be surrounded with the #[and #] special characters to put these references into a single environment, to assure they all refer to the "same" variable.

[7] Future functionality

Chapter 3

Tutorial — Builtins

3.1 When to use a Builtin

If a lispfn isn't sufficient, either because you want to handle (unbound) variables in your arglist, or want to declare side-effects, you need to use a builtin. The interface of a builtin to the reasoner is substantially different from a lispfn. A lispfn takes the arguments it expects to have as if the rhet form were a lisp function application, and returns a non-nil result to indicate success. A builtin takes two additional arguments before the other arguments it would expect from the form: a failure continuation and a continuation for variables that were protected before calling the builtin (more on protection in a second). A builtin can do one of two things. It can invoke the failure continuation if there is no possible solution, or it can returns a generator which will return solutions each time it is called until there are no solutions left, and then it will invoke the failure continuation. This generator is expected to be a function of one argument. The argument is the so-called culprit. If it is NIL, the next proof should simply be supplied. Non-nil culprits will be covered under the *Writing More Advanced Builtins* section below. It is possible, however, to always ignore the culprit and merely generate the next possible proof. The culprit is used for the failure-driven backtracking mechanism which will be described in more detail below.

Generators are lexical closures in common lisp. Normally the failure continuation, for example, are passed via the lexical environment to the closure. Remember the closure will only be invoked with one argument so it is important to have everything else you may need in the generator's lexical environment.

Several macros have been written to make writing a builtin more straightforward.

3.2 Writing Simple Builtins

Lets start by looking at two simple builtins. The first one will be completely deterministic[1]. This is still more powerful than a lispfn, because it can still bind a variable. But being deterministic, it must either succeed or fail: it will not backtrack.

(3.1)
```
(ADD-INITIALIZATION "Define builtin:  HFUNCTION-VALUE"
    '(DEFINE-BUILTIN 'HFUNCTION-VALUE "FUNCTION-VALUE"
                     '((OR FORM FN-TERM) T) :NF-NM)
    () 'USER::*RHET-REPEATABLE-INITIALIZATIONS*)
```

(3.2)
```
(DEFUN HFUNCTION-VALUE (FAILURE REINVOKE TERM VALUE)
    "Succeed if the value of the function term Term
is (unifies with) Value"
    (DECLARE (TYPE CONTINUATION FAILURE REINVOKE)
             (TYPE ARBFORM TERM)
             (TYPE T VALUE)
             (OPTIMIZE SAFETY))

    ;; this is strictly a lookup function and is side-effect free.
    ;; Since the form can have at most one value, it is also
    ;; deterministic.

    (RATIONALIZE-ARGUMENT TERM :DO-FORMS T)
    (RATIONALIZE-ARGUMENT VALUE :DO-FORMS T)

    ;; react somewhat differently depending on where the rvariable
    ;; is (or if there is one).

    (TYPECASE VALUE
       (RVARIABLE
        (UNIFY-VAR FAILURE REINVOKE VALUE (FUNCTION-VALUE TERM)))
       (OTHERWISE
        (BUILTIN-EXECUTES-ONCE-ONLY FAILURE REINVOKE T
            (COMPLEX-UNIFY (FUNCTION-VALUE TERM) VALUE)))))
```

Note the usage of the two parameters before the form that is used to call the builtin; these are standard for all builtins. We first use the macros E-Unify:Rationalize-Argument to handle the arguments to the builtin that will come from the Rhet form being proved. This macro makes sure that when we reference one of our arguments, we get what the variable was bound to and not the variable itself if it is bound. In general this means that we cannot

[1] This example is obsolete functionality as of version 16 (that is, the Function-Value builtin), but it is still illustrative.

take advantage of intelligent backtracking (since we won't be able to identify the culprit, if it is a poor choice of binding for one of our variables) but builtins rarely take advantage of this anyway since the cost to compute the culprit is often too high.

Here is another builtin that is nondeterministic (it can backtrack):

(3.3)
```
(ADD-INITIALIZATION "Define builtin:  HGENVALUE"
    '(DEFINE-BUILTIN 'HGENVALUE "GENVALUE"
                     '((OR RVARIABLE LIST) T-LIST) :NF-NM)
    () 'USER::*RHET-REPEATABLE-INITIALIZATIONS*)
```

(3.4)
```
(DEFUN HGENVALUE (FAILURE IGNORE RVARIABLE* LISP-EXPRESSION)
    "Sets the Rhetoric rvariables to the first value in the lists
    returned by evaluating the lisp-expression.  Other values are
    used for backtracking.  This is like the current definition, but
    if the lisp expression returns multiple values, it will bind them
    to each rvariable in turn, the car of each value returned.
    Additional values beyond the number of rvariables supplied
    are ignored.  Additional cdrs of the lists returned are used for
    backtracking.  Should one list be shorter than another, it will
    supply a value of NIL if backtracking proceeds to that point."
    (DECLARE (TYPE LIST RVARIABLE* LISP-EXPRESSION)
             (TYPE CONTINUATION FAILURE)
             (OPTIMIZE SAFETY))

    ;; This functions lisp-expression argument will be a list, not a
    ;; real function (otherwise fiddling with it's arglist for those
    ;; terms that are bound rvariables would be hard!)  At this point,
    ;; the vars should be bound, or the lisp function may get a
    ;; surprise!  (sorry, not my problem!!!)

    (IF (NOT (CONSP RVARIABLE*))
        (SETQ RVARIABLE* (LIST RVARIABLE*)))   ;make it a list.

    (ASSERT (EVERY #'RVARIABLE-P RVARIABLE*) ()
            "Not all the forms passed to GENVALUE
            in the Rvariables position are rvariables")

    (LET ((RESULTS (MULTIPLE-VALUE-LIST (EVAL LISP-EXPRESSION))))
      #'(LAMBDA (CULPRIT)
            (DECLARE (TYPE CULPRIT-TYPE CULPRIT))

            (IGNORE CULPRIT) ;not used.  Could be, though.
            (LET ((FAIL T))
               (COND
```

```
((EVERY #'NULL RESULTS)
 (INVOKE-CONTINUATION FAILURE)) ;done.
(T
;; generate the next proof of the goal.  The
;; interpreter doesn't cache (yet, anyway) to
;; make debugging easier.
 (WHILE FAIL
    (SETQ FAIL NIL)
    (SETQ RESULTS
          (MAPCAR
             #'(LAMBDA (X Y)
                  (CHECK-TYPE Y LIST)
                  (UNLESS
                     (UNIFY-RVARIABLE X (CAR Y))
                     ;; this one fails,
                     ;; finish taking cdrs,
                     ;; then try next
                     (SETQ FAIL T))
                  (CDR Y))
             RVARIABLE*
             RESULTS)))
 :GENVALUE-RESULT))))) ; ''Justification''
```

The important thing to note here is that the result of the builtin is a CLOSURE. This closure is designed to be called and return a new binding (as a side effect) of the variable passed to the builtin. It does this with the E-Unify:Unify-Rvariable function. It returns a justification structure, which in this case is just a keyword[2].

It is important that when the generator exits it pass back some non-NIL result. The reasoner has no way of noticing the side effect of destructive binding of some variable. A NIL justification, then, is interpreted as a failure.

3.3 Writing More Advanced Builtins

The main things this section will discuss are

- What a Builtin may have to know about how the interpreter and compiler works (more advanced control strategies).
- How to identify a culprit.
- How to handle a culprit identified elsewhere.

[2]Normally one would return a list of the fact or axiom structures needed to cause the builtin to succeed, but in this case since we are dependant on an ephemeral call to some arbitrary lisp function, we can't do so.

If you find the above necessary for the builtin you wish to construct let me add one cautionary note: there is no substitute for READING THE SOURCE CODE. Find a builtin that already does pretty much what you want, and understand it thouroughly.

3.3.1 Culprits

Culprits are from the literature on intelligent backtracking [Warren, 1986] [Bruynooghe and Pereira, 1984] [Cox, 1984]. The idea is that all functions that can backtrack take a failure continuation as an argument. If they cannot find any proof for the form handed them, they invoke the continuation. If they can identify a particular *culprit*, however, a slot on the rvariable contains the continuation that should be invoked. For example, given the following: `[[P ?x] < [Q ?y] [R ?x ?y]]` if we use naive backtracking, given the goal `[P A]` if we can prove `[Q B]` we will try to prove `[R A B]`. If that cannot be proven (possibly after much work), we will backtrack and reprove `[Q ?y]` for some other value, possibly `[Q C]`. Intelligent backtracking allows us more latitude: if during the course of proving `[R A B]` we determine that `[R A ?z]` would have failed (that is there is no proof of predicate R with it's first argument bound to A) we call ?x the *culprit*. Since in our rule ?x is a local, it does no good to reprove anything else in the rule, the rule can immediately fail. Should the rule have been *deterministic*, that is, the "last" possible way to prove `[P A]`[3] then we can in fact invoke the last backtrack point... possibly all the way up the stack to the point were A was bound. A more complete example: Let's say that this was our only rule to prove P, and it was invoked as a subgoal of the rule: `[[Z ?x] < [P ?x] [T ?x]]`; further that is the only way to prove Z; and it was invoked by the rule `[[GOAL ?x] < [R ?x ?y] [Z ?y]]`. `[GOAL T]` was the thing the user asked the system to prove, and the KB has `[R T A]`, `[R T C]`, `[R C F]` and `[Q B] [Q F]` in it. This binding of ?y to A is the culprit by our logic above, so once we determine that we will immediately retry that subgoal, getting the new binding of C.

3.3.2 Continuations

Continuations are used much as they are in Scheme [Abelson and Sussman, 1985][Rees and Clinger, 1986], that is, in order to restore a particular dynamic environment. Since Common Lisp does not support continuation passing, however, a special-case must be painfully emulated, since these continuations must be passed up as well as down the stack. In particular, Rhet keeps track of enough information in order to restore the proof tree to any particular state, and continue from that particular point. In sketch, the following occurs:

- A goal is attempted to be proved. Rhet sets up a continuation structure, remembering it on Reasoner:*Current-Continuation*, and Reasoner:*Root-Continuation* if this goal is from the top-level.

[3] either it is the only rule for proving `[P A]` or all other rules to prove it have failed

- Reasoner:Prove-Based-On-Mode is invoked, which will first attempt to look up the goal in the KB directly, then invoking BC on any axioms that match. The proof strategy in implemented within this routine. First it
- creates a continuation which it will use to attempt further proofs.
- It then calls Reasoner:RKB-Lookup to find the goal in the KB. If RKB-Lookup fails, it will invoke our continuation, but if it succeeds we can return it (or collect it, if we are doing a Query:Prove-All).
- When our continuation is invoked, and the last time RKB-Lookup was invoked it returned something useful, we will reinvoke it. When RKB-Lookup runs out of proofs
- we invoke BC by calling Reasoner:Generate-BC-Proof. This essentially is a reinvocation of our first step, above.

Note that any time we bind a variable, the continuation that is in *Current-Continuation* is stuffed there too, so we know what continuation was current at that time.

Each time we expand the stack, we establish a new continuation, and set up a catch for it. Establishing a new continuation also means that we identify it as either a child (a subproof of the parent) or on the same level as the last continuation. Rhet then tracks all the continuation structures by arranging them in a tree in parallel to the implicit proof tree. This continuation tree is an explicit representation of the entire proof tree. unlike the stack which is a dynamic instance of the active part of the proof tree. Now the trick is that when a culprit is identified, we will want to reinvoke that part of the proof tree, identified by the continuation, that will cause a new binding of the variable.

In the simple case, when no culprit is identified, we will invoke the continuation associated with the last binding of a variable. Invoking a continuation consists of determining what flag to throw to, and passing an appropriate culprit. If we cannot identify a particular culprit, we can either pass a set of them, or the variable last bound. If this continuation appears above us on the stack, it is simply thrown to, and the associated generator is then invoked to give an alternate binding to our variable. If the continuation is somewhere to the "left" of us in the proof tree, then we must first reexpand the stack out to the point when the continuation was in force, and then reinvoke the generator. The support macros listed below automate this process. Basically, given a culprit, they can determine if the continuation is later than their own, which implies it is either to their right or below them. Since they would not have been reinvoked it it were to the right of them, they use Rllib:Repeat-Invoke-BC to reestablish their subtree originally established with Rllib:Invoke-BC.

At each level, then, the culprit's continuation can be compared to any established at the current level of the proof, and appropriate action taken. Ultimately, the appropriate generator is reinvoked, a new binding is established for the variable, all variables with later continuations are unbound, and the proof proceeds from that point as afresh, generating a new proof tree from that point. This can be taken to be an implemenation of an "Oracle"[Hopcroft and D., 1979], since from the program's point of view the subsequent

computation has no longer taken place, it simply has been "advised" that the chosen binding is a poor one.

3.4 Language Constructs

UI:*Builtin-Trigger-Exception-List* This is a list of the BuiltIns that *are* valid trigger candidates. All others are rejected.

Rllib:Define-Builtin *Symbol User-Name Argument-Type-List Builtin-Type* &Optional *Assert-FN-Symbol Undo-Compiled-Function*
This function defines Symbol to be a builtin, and optionally associates a user-defined function to process undo requests (which otherwise are ignored). The function, like all builtins, is expected to return a generator. Not supplying an Undo function to Define-Builtin is defining the function as side-effect free. Builtins with side effects that cannot be undone are encouraged to supply a Undo function that sends a warning to the user. The Argument-Type-List is a user-interface specified list of the types of the arguments the builtin expects, as a hint to the parser, and to allow the high level user interface to catch syntax errors. The Builtin-Type is a keyword which indicates to Rhet the kind of builtin it is, for compilation or constraint purposes, *e.g.* foldable, non-monotonic. The User-Name is how the builtin will be known to the user and should be a string. The optional Assert-FN-Symbol should be the symbol for a function to be called if this builtin is asserted, as in appearing in the LHS of an FC axiom, or as an argument to Assert:Assert-Axioms.

E-Unify:Rationalize-Argument *Thing* &Key *Do-Forms*
Call this on args to builtins to change the vars to be their bindings, and, if Do-Forms is non-nil, forms to be simplified. Otherwise the argument is left alone.

E-Unify:Real-Reference *Thing*
If thing is a variable, return it's binding, otherwise Thing itself.

Calling any of the below functions will return a closure which will do the indicated operation. All functions take a non-optional failure-continuation parameter for passing the continuation to be taken if the function cannot succeed (and cannot identify a culprit). This closure is then normally called immediately to attempt the indicated function. Successive calls will generate additional possibilities as appropriate. This closure is returned as the second value from a builtin: some builtins without backtracking possibilities will simply return their computed value and a second value of nil to save the work of Consing up a closure.

Note that parameters may be preprocessed by the interpreter, but rvariables should not be. This is because the intelligent backtracking capabilities rely on knowing where a rvariable was bound if a bound rvariable cannot be unified with anything to prove the subgoal. Each of the following functions exist in two forms. For a function Frotz, calling Frotz does the indicated operation. Normally only the interpreter will make such calls.

Calling Frotz-mac (the macro form) will expand into Lisp code that can be compiled by the axiom compiler.

In the documentation below, the failure continuation parameter, which is always the first parameter, and the second value returning the closure are omitted.

First, here is a list of the macros this library defines which are meant to be used to appropriately manipulate the closures, failure continuations, and what they return. For a good example of the calling sequence, look at the axiom interpreter functions, *i.e.* Reasoner:Interpret-FC-Axiom and Reasoner:Interpret-BC-Axiom. The former is simpler, and should be examined first.

Reasoner:Create-Continuation *Child-or-Right &Optional* (Name-String "Cont-") (Current-Continuation *Current-Continuation*)
Create a continuation structure.

Reasoner:*Current-Continuation* Bound to the continuation structure that is current for this proof level. Used to update variable's Where-Bound field when bound.

Reasoner:*Root-Continuation* Bound to the top level most general continuation from the top-level proof.

Rllib:Define-Continuation *Continuation* &Body *Body*
Sets up a continuation at this point, binding *Current-Continuation*. Only code within the body of the Define-Continuation may invoke it.

Rllib:Invoke-Continuation *Continuation* &Optional *Culprit*
Invokes a continuation. Anything on the stack above the continuation point is lost. Note that continuations are expected to accept a Culprit argument[4].

Rllib:Debug-Continuation *Continuation* &Optional *(Msg "Establishing Continuation of ") Pro*
This is used in the other continuation macros to simplify debugging. Pro contains the protected rvariables, if there are any.

Rllib:Create-Generator *GenVar Creator*
Uses Creator to create a generator, assigning it to Genvar.

Rllib:Invoke-Axiom-With-Failure-Continuation *Axiom* &Rest *Arglist*
Invokes the Axiom on the passed Arglist but adds an additional (first) argument: a continuation point that on failure will return to the code following this macro.

Rllib:Invoke-Generator *Generator* &Optional *Culprit*
Builtins, *etc.* return generators (closures) which need to be invoked to generate "proofs". This macro does the work. It passes a Culprit, which can be NIL if no culprit is known[5].

[4] Actually, this is a function, to aid in debugging; eventually it should be declared inline.

[5] As with Rllib:Invoke-Continuation, this is actually a function to aid in debugging, and should eventually be declared inline

Rllib:Invoke-Generator-With-Failure-Code *(Generator Culprit)* &Body *Failure-Code*
This function is like Rllib:Invoke-Generator except that if the Generator fails, the Failure-Code is run before the macro returns.

Rllib:Invoke-Deterministic-Builtin *Builtin* &Rest *Arglist*
Invokes the passed builtin (passed as a symbol) on the passed arglist, but adds an additional (first) argument: a continuation point that will cause continuation at this level (invoked on failure). Returns whatever the builtin does.

Rllib:Invoke-Non-Deterministic-Builtin *(Builtin* &Rest *Arglist)* &Body *Success-Code*
Invokes the passed builtin (passed as a symbol) on the passed arglist, but adds an additional (first) argument: a continuation point that will cause continuation at this level (invoked on failure). Returns whatever the builtin does. This is similar to Rllib:Invoke-Deterministic-Builtin, but on success, the Success-Code is run. Note that locals available inside of Rllib:Invoke-BC-Protecting-Unbound-Globals are also available in the Success-Code.

Rllib:Invoke-FC *((Function* &Rest *Arglist) Backtrack)* &Body *Success-Code*
This calls the function on the arglist passing two extra (first) arguments: an appropriate continuation point on failure, (that cannot be traced to a rvariable) and the continuation point set up for this level, used to identify which rvariables the function may have bound internally. The function will return a generator which can be used to get things appropriately bound in the globals-list for what constitutes a proof. Unbound globals are given a where-bound value of the continuation this macro generates. The idea is that if that rvariable later turns out to be a culprit, we can non-locally get back to the continuation generated by this macro. If the Func cannot generate *any* proofs, we invoke the passed Backtrack-Point. If we do generate some proof, control passes to the body of the macro. We bind local rvariables that are valid within the body: CONTINUATION which is the continuation we generate, GENERATOR which is the generator the Func returns, and *Result which is what the generated returned on invocation. After the Success code, we try the next value. Eventually we fail and invoke the Backtrack-Point. If the Backtrack-Point is NIL, we fall through.

Rllib:Invoke-BC *((Function* &Rest *Arglist) Deterministic Backtrack-Point* &Key *(Generator '*Next-Value) (Continuation '*This-Failure-Level) (Continuation-Type :Right))* &Body *Success-Code*
Unsurprisingly similar to Rllib:Invoke-FC-Protecting-Unbound-Globals in effect, though unfortunately dissimilar in execution. Mainly, this is due to FC and BC using the stack differently. BC returns a value, then must futz around reexpanding the stack to return another value, and FC doesn't need to return anything, so it is more efficient: when the stack is most expanded, it will add it's chained form.

Rllib:Repeat-Invoke-BC *(Generator Culprit This-Failure-Level* &Key *(Failure-Code NIL))* &Body *Success-Code*
Very similar to Rllib:Invoke-BC-Protecting-Unbound-Globals, except that macro is to initially do a proof, while this one is to get a subsequent value given an existing

generator, and passing a Culprit. Virtually all of the arguments to this function are things that should have been saved from calling that one.

Rllib:Barf-On-Culprits *Culprit-List Jumpstart*

Given a culprit, this function does the right thing to unwind the system to the state it was in at assignment to the culprit (thus it will potentially get another value). The jumpstart continuation is used when the culprit is below us in the 'stack', this invokes the parent which will reinvoke us.

Chapter 4

Dictionary of Useful Functions

Note that while we are documenting many internal functions here which are useful or needed to write Lispfns and Builtins for Rhet, we make no claims to the stability of these functions between releases, or even patchs. That is, unlike the things that are documented in the User's Manual, these are really *internal* functions, and therefore subject to change with minimal notice.

These functions listed here are not meant to be exhaustive, but merely the set of funcitons from the reasoner and other packages that are most likely to be immediately useful when writing a builtin or lispfn. If, for example, your builtin needs to use the unifier directly, you should refer to the unifier chapter of this manual for more information about calling the unifier: it will not be listed here.

E-Unify:Bound-Var-In-Goal-P *Form* &Key *Bound-In*
A predicate that returns non-nil if the goal (usually a form) has any rvariables bound. If Bound-In is supplied, only rvariables with that as a Where-Bound field are candidates.

E-Unify:Clear-Binding *Rvariable* &Optional *Constraints Maintain-Continuation*
Resets the binding and optionally the constraints of the passed Rvariable to NIL.

E-Unify:Clear-Some-Bindings *Term Binding-Location* &Key (Test #'Continuation->=)
Unbind the rvariable, or if Term contains rvariables, unbind them if it (they) is (are) bound at the Binding-Location, as well as any constraints asserted at the Binding-Location. It returns the Term again.

E-Unify:Constrain-Term *Term-to-Constrain Constraint-Form*
This function constrains the rvariables in the first argument such that the Constraint-Form holds.

E-Unify:Continuation-> *Cont1 Cont2*
Returns non-nil if the first continuation is 'greater' (occurs later) than the second.

E-Unify:Continuation-= *Cont1 Cont2*
Returns non-nil if the continuations are equivalent

E-Unify:Continuation->= *Cont1 Cont2*
Returns non-nil if the first continuation occurs later or at the same (stack) level as the second.

E-Unify:Convert-Form-to-Fact *Form* &Key *Context Truth-Value*
This takes a form without rvariables, and for any subforms assigns canonical names to them (so references will work). If the fact is found in the KB it is returned, as well as a second value of T. Otherwise we return a constructed fact and Add-Fact is NOT called on this new fact. Normally that will be the first thing called after this call. The Truth-Value defaults to the truth-value of the form being converted, and can be overridden with the keyword option.

E-Unify:Convert-to-FN-Term *Term*
This takes a term without rvariables, and for any subforms assigns canonical names to them (so references will work) and then returns either a new or found fn-term that is equal to the argument. The second value returned indicates if this term were just added to the KB, if NIL, it was found there.

E-Unify:Crunch-Vars *Term*
The Term is returned with any bound rvariables fully expanded. It takes pains to return a form that is EQ to the passed form if nothing has changed, and a copy if something has changed so that the argument is not destroyed.

E-Unify:Term-Unifies-With-Form-P *Term Form Context &Key Recursive*
Returns non-nil if the term and form E-Unify in the passed context. The second value is a list of variables bound in Term to allow the unification.

E-Unify:Get-Binding *Rvariable*
Returns the current binding of the passed Rvariable.

E-Unify:Last-Bound-Vars *VAR-LIST*
Given a list of rvariables, return the set of those rvariables most recently bound. Normally this will be used to determine a culprit if better info isn't available, the last bound var is the one to invoke.

E-Unify:Simplify-Form *Form &Optional Context* &Key *As-Fact*
The most simplified form (*e.g.* with canonical names substituted as appropriate) is returned. If Form has any rvariables, an error is signaled. If the Form can be resolved to a canonical name or fact, it is returned. Note that the fact returned is not necessarily appropriate for adding via add-fact, since canonical names may appear in it's arglist. Use Convert-Form-To-Fact instead. If As-Fact is specified and non-Nil, then Facts are prevented from being coerced into Canonical Names for the return value, since certain callers prefer the Canonical Name.

E-Unify:Type-Restrict-Term *Assertable-Term Type-Struct Context*

Updates the type of the passed Term to be type Type-Struct (an Itype-Struct) in context Context. On canonicals, each memeber of the class will be updated. The new term is returned. In case of error, :TR-TERM-ERROR will be thrown.

E-Unify:Unbound-Vars-In-Term *Term*

Returns a list of all 'unbound' vars in the passed term. This includes vars that are bound at a level on the stack 'above' the current one. It also includes unbound variables in the bindings of the bound ones!

RAX:Copy-Axiom *Axiom*

Since variables are destructively bound, this is the canonical way to assure a unique copy.

RAX:Freeze-Axiom *Axiom*

Takes a (BC) axiom and 'freezes' it. This will turn rvariables, *etc.* into unique symbols, s.t. the resulting list can be operated on like a Lisp list. Thus `[[A ?y] < [B ?x ?y]]` which is an axiom structure will become ((A *VAR*-y-123) < (B *VAR*-x-243 *VAR*-y-123)) where each are atoms in a list rather than structures.

RAX:Thaw-Axiom *Frozen-Axiom*

Takes a frozen axiom (as produced by RAX:Freeze-Axiom) and thaws it: *i.e.* generates a RAX:BC-Axiom structure from it. The rvariables are guaranteed to be unique, so (Thaw-Axiom (Freeze-Axiom Axiom-Foo)) can be used as a mechanism for structure copies.[1]

RE-to-DFA:Compatiblep *String DFA*

Compatiblep Returns T if the passed DFA, from RE-to-DFA:Convert-RE-To-DFA, could have generated the String.

RE-to-DFA:Convert-RE-To-DFA *String*

This function converts REs to compiled DFAs. The RE is passed as the the String parameter.

Reasoner:Abort-Rhet *NIL*

Abort Backward Chaining. Possibly needs to clear up FC too...

Reasoner:Chain *Fact &Key Context*

Called by the RLLIB package when a fact has been added by the user. This invokes the forward-chaining mechanism as needed. It may call itself recursively, or as the result of an Hassert-Axioms form. Note that memory used by FC goes in it's own area.

Reasoner:Constraint-Satisfy-P *Constrained-Rvariable Form*

Returns T if setting the constrained-rvariable to the Form is not a violation of the rvariable's constraints.

---

[1] In fact, this is the idiomatic way to do so.

Reasoner:Disprove-Goal *Legal-Goal Proof-Type* &Key *Fail-if-Builtin Succeed-if-Builtin*
Returns a proof of `[NOT GOAL]`, short-circuiting depending on keyword options.

Reasoner:Generate-BC-Proof *Failure-Continuation Reinvoke-Continuation Arbitrary-Form*
For practical purposes, a Builtin. Returns a generator to prove a goal that internally handles the generators of individual clauses defined that unify with the goal.In order for cuts and such like to work, this is the "builtin" that is the interface to the BC axioms. This is to assure that whatever set of causes indexing may proscribe as appropriate for the current proof, the CUT form can interact with this code (stackwise) and do the appropriate cutting action. Also, this makes prove-based-on-mode a *little* simpler...rather than having to do the indexing and get the generators for the axioms itself, this routine returns one generator (which internally will switch between the appropriate indexed generators). Note that in a proof tree, this routine represents an OR node: no state needs to be saved from prior things invoked; no caching is done at this level (only axioms, not builtins, can cache).

Reasoner:Interpret-BC-Axiom *Failure-Continuation Reinvoke-Continuation Axiom Goal*
Simulates the Axiom using the passed Rvariable bindings. This will recursively call the reasoner to prove any subgoals of Axiom. If successful, it returns a first value of T and a second value of the new rvariable bindings. If unsuccessful, it's first value is Nil. This function is used by the reasoner to handle interpreted (vs. compiled) BC axioms. This is a prerequisite to using the stepper, and certain advanced trace facilities. All work is done in the Rhet-Terms:*Current-Context*.

Reasoner:Interpret-Builtin *Failure-Continuation Reinvoke-Continuation Form*
This function interprets a single clause in an axiom, which has already been determined to be a builtin. It returns whatever the builtin would return, normally 1. success or failure, and 2. the closure, if needed. All work is done in the Rhet-Terms:*Current-Context*.

Reasoner:Interpret-FC-Axiom *Failure-Continuation Axiom Trigger*
Simulates the Axiom using the passed Trigger. This will recursively call the reasoner only if a Prove form is encountered, to prove those subgoals of Axiom. If successful, it returns a first value of T and a second value of the rvariable bindings used. If unsuccessful, it's first value is Nil. This function is used by the reasoner to handle interpreted (vs. compiled) FC axioms. This is a prerequisite to using the stepper, and certain advanced trace facilities. All work is done in the Rhet-Terms:*Current-Context*.

Reasoner:Interpret-Lispfn *Failure-Continuation Reinvoke-Continuation Form*
This function interprets a single clause in an axiom, which has already been determined to be a user supplied Lisp function. It returns whatever the Lisp function returns. All work is done in the Rhet-Terms:*Current-Context*.

Reaonser:Lookup-Lispfn *Name*

Returns multiple values, the first of which is the argument symbol (to make lookup a predicate on a declaration for lisp-function-ness) the second is the Predicate-Function and the third of which is the Assert-Function as declared for function name by a Rllib:Declare-Lispfn form. If no such form has been supplied for name, all values are Nil. Note that if the Assert-Function returned is Nil it means it was not supplied to the Declare-Lispfn form, and so the normal assertion mechanism should be used. The fourth value is a list of types that the arguments are expected to be subtypes of, for runtime typechecking purposes.

Reasoner:Prove-Based-on-Mode *Failure-Continuation Reinvoke-Continuation Goal*

This does a prove simple or prove complex (*etc.*) depending on the call made originally by the user. It should be the common re-entry point to the reasoner for recursive proofs (*i.e.* subgoals). It returns a generator which is used to get the actual proofs. (Generator will return justifications for each proof). It uses Reasoner:*FC-Active* to tell if it is being invoked from FC, and if so, will not try to find BC axioms that prove the Goal.

Reasoner:RKB-Lookup *Failure-Continuation Reinvoke-Continuation Arbitrary-Form* &Key *Context*

This function returns a closure that attempts to find something matching the arbitrary-form in the KB. If it succeeds, it returns a fact, passed rvariables are (destructively) bound as a side effect. If it fails, it invokes the FAILURE continuation, unless it identifies a culprit. The Unbound-Continuation is what others will use to reinvoke me if necessary.

Rhet-Terms:Accessible-HN *Itype-Struct* &Key *Head Hash-Index Defaultp Index Truth-Value Context*

This will return all facts that are accessible to the current (or specified) Context.

Rhet-Terms:Archive-and-Return *Function Args Type Result*

Do the work of dibbling a call to Function on Args with Result. Type is either :assert or :query, and indicates the type of function being called (only one may be recorded, as per Assert:Rhet-Dribble-Start.

Rhet-Terms:Copy-Goal *Goal &Optional Keep-Scratchpad*

Since variables are destructively bound, this is the canonical way to assure a unique copy.

Rhet-Terms:Create-Form *Value* &Optional (Truth-Value :True)

A standard fn for building forms, which will update the form-rvariables field. It is a useful interface to Rhet-Terms:Make-Form. To convert lisp atoms directly to a form structure, use UI:Cons-Rhet-Form instead.

Rhet-Terms:Create-Rvariable *Pretty-Name* &Optional (Type *T-U-Itype-Struct*)

A function to construct variables. The Pretty-name is a string that will be the printed appearance of the variable. As such, it should begin with the character "?".

Rhet-Terms:Find-Fact *Head Arglist* &Key *Context Truth-Value*
This returns the interned fact if it can be found. As a second value it returns the type (as an Itype-Struct structure), and as a third value, it returns the truth value (so it need not be looked up). Note that the arglist can only consist of facts or canonical-names, function terms must be converted to canonical names. This function will look find a fact accessible to the passed context, rather than strictly in the passed context[2].

Rhet-Terms:Find-FN-Term *Head Arglist* &Key *Context*
This returns the interned function term if it can be found. As a second value it returns the type (as an Itype-Struct structure). Note that the arglist can only consist of facts or canonical-names, since function terms must be converted to canonical names.

Rhet-Terms:Find-Rvariables *Term*
Calculates the Rvariables field for a term by finding any rvariables and returning an list of the rvariables mentioned. Normally, one would use, *e.g.*, Rhet-Terms:Create-Form which returns a form with this calculation made.

Rhet-Terms:Find-or-Create-Term *Head* &Optional *Arglist* (Type *T-U-Itype-Struct*)
If the appropriate function term is already known, it is returned. Otherwise, it is created and interned.

Rhet-Terms:Freeze-Goal *Goal*
Takes a legal goal and freezes it. Interface to Rhet-Terms:Freeze-LFP.

Rhet-Terms:Get-Canonical *EQ-Term Context* &Key *Local Anything-OK*
This function takes a Term and a Context and returns the canonical name for the term in the context, or Nil if none exists. If the Local argument is non-nil, treat not having a canonical name directly in the Context the same as not having one at all.

Rhet-Terms:Get-Frame *Type-Symbol*
Looks up the type's frame definition and returns the frame (an object of type Rhet-Terms:REP-Struct).

Rhet-Terms:Get-Frame-from-Type-Hack *Type-Symbol*
Like Rhet-Terms:Get-Frame, above, but since the user can give us a keyword or list for the type, and internally we want itypes, this returns the frame either way.

Rhet-Terms:Get-Predicate *Thing*
Given a Fact, or a Form, this function returns the purported predicate, *i.e.* the atom that is the head. Thus for Thing: fact [F A B] whose head is 'F and form [F A B] whose head is a FN-Term for a Fact whose head is F and args are Nil; both would return the symbol F from this function. Canonical Names will use their primary. Any other argument generates an error. We take Canonical Names and even FN-Terms with Canonical Names so we can be called inside of certain other functions, though they are not strictly predicates.

[2]It is possible for Find-Fact to return a canonical name, for example, if the passed arglist consists only of canonical names, it is likely that a canonical name is interned in the hashtable.

Rhet-Terms:Get-Result-Itype-Struct *Function-Atom List-Of-Itype-Structs &Key Inhibit-Maxtype*

The second argument should be a list of Itype-Struct structures representing the types of arguments to function-atom. The most specific type inferable of 'function-atom (arg1 ...argn)' where each arg has the Itype-Struct specified by Argument-Itype-Struct-List will be returned. In particular if Argument-Itype-Struct-List represents invalid types for function-atom, Rhet-Terms:*T-Nil-Itype-Struct* is returned.

The two primary ways Unifier can use 'Get-Result-Itype-Struct' are the following.

1. To see if [Any ?X*Type1 [F ?X]*Type2] unifies with constant [A], where VAR is a lisp variable whose value is the (constrained) rvariable structure, and A is the fn-term for the constant...

   After calling

   (Typecheck (Get-Type [A]) (Get-Type VAR))

   the Unifier should call

   (Typecheck (Get-Result-Itype-Struct 'F (Get-Type [A])) (Rhet-Term-Type (Car (Var-Constraints VAR))))

   noting that (Car (Var-Constraints VAR)) is an oversimplified way of getting the Rhet-Terms:Itype-Struct structure of the first constraint on VAR.

2. To see if ?X*Type1 (call it VAR) and [F ?Y*Type2] (call it FORM1) can unify, the Unifier should call

   (Type-Exclusivep (Get-Type VAR) (Get-Result-Itype-Struct 'F (Get-Type (Form-Rvariables FORM1))))

   (which is again, an oversimplified example of how one would access the type of embedded rvariables in a form), just as the Unifier calls

   (Type-Exclusivep (Get-Type VAR) (Get-Type (Form-Rvariables FORM1)))

   to see if ?X*Type1 and ?Y*Type2 can unify. If T is returned, the Unifier can detect failure. Otherwise, the Unifier can post constraints by producing [Any ?Y*Type2 [F ?Y]*Type1] by calling (given VAR2 is ?Y*Type2):

   (Constrain-Var VAR2 (Create-Form 'RLLIB:HTYPE-QUERY (LIST FORM1 (Get-Type VAR))))

   Note that in general, the resultant type may be a subtype of both input types — the Rhet unifier uses TypeCompat and sets the type of both objects appropriately.

Rhet-Terms:Get-Type *Arbform*

Gets the type of an arbitrary object and returns it.

Rhet-Terms:Log-to-Archive *String*

Puts the string out on the archive (if it is active) as a comment.

Rhet-Terms:Make-I-Type *<Type Symbol or List>* &Optional *Permissive*

Converts the passed lisp symbol into an Rhet-Terms:Itype-Struct. If a list is passed, it is treated as a type specification as would appear after the astrick on a variable. This

function is typically used, for example, by builtins that expect an argument to be a type specification. If Permissive is NIL, the default, return NIL if any component type is undeclared.

Rhet-Terms:Rhet-Equalp *Term1 Term2*
Returns non-nil if the two things are CL:EQUALP or Rhet terms that are equivalent (not in the unification sense). If we (Copy-Goal [Foo ...]) the result should be Rhet-Equalp to the original.

Rhet-Terms:Set-Argument-Itype-Struct *Form* &Optional (Result-Itype (Rhet-Term-Type Form))
Given a form and a desired resultant type, updates the types of arguments as necessary to guarentee a result that is that type (or a subtype of it). It returns the destrutively modified form if something was changed.

Rhet-Terms:Thaw-Goal *Frozen-Goal*
Takes a frozen goal and thaws it. Essentially an interface to Rhet-Terms:Thaw-LFP.

Rhet-Terms:Update-Type *Term* &Optional *Force-Type &Key Non-Heuristic For-Equality*
Destructively modify the passed form with a new type calculated from the Form's head and arguments, unless Force-Type is set, in which case use that and constrain the subforms appropriately. Be intelligent about forms that you can't calculate. Returns the term, and a list of vars constrained.

Rhet-Terms:Warn-or-Error *Item Checklist-symbol Continue-Control-String Proceed-Control-String Format-Control-String* &Rest *Format-Args*
Like Warn, returns non-nil if the user wishes to continue. A nil return implies failure; the calling builtin or lispfn would normally invoke a failure continuation. A bit snazzier than just WARN or CERROR, this function does a warning, and then asks if the user wants to go ahead (continue-control-string), error out, or go ahead and not ask again. Item and checklist is supplied by the caller for just this functionality: if item is found on checklist, Warn-or-Error will return non-nil; the user had previously indicated they wanted to Proceed on this sort of error. Checklist symbol is passed as a symbol; the standard place to put simple things (where the Item is a hardcoded keyword) would be on the Rhet-Terms:*General-Warning-List*, however the user is free to Defvar their own Checklist. They should then add an initialization to clear it to the Rhet-Terms:*Warn-or-Error-Cleanup-Initializations* initialization list.

Rllib:Barf-On-Culprits *Culprit-List*
Given a culprit, this function does the right thing to unwind the system to the state it was in at assignment to the culprit (thus it will potentially get another value).

Rllib:Builtinp *Symbol*
Returns five values: The first is non-nil if the symbol is the name of a builtin predicate; it is in fact a string which is the name of the builtin as the user would call it from

within Rhet. The second value is a list of the types of the arguments to this builtin (see Rllib:Define-Builtin. The third is a keyword description of the type of the builtin, e.g. is it monotonic, foldable, *etc.*. The fourth is the symbol name of a function to be called if this builtin is asserted, as in the LHS of a FC axiom. The fifth is an UNDO function that is called to undo calls to a builtin with side effects.

Rllib:Complement-Truth-Value *Form-To-Complement*

This function takes a form and inverts the expected truth value. This is used by the prover to decide if it should look at facts like X or `[NOT X]` which is represented by the truth value on the fact. The form returned is EQ to the form passed, so the function is DESTRUCTIVE!

Rllib:Generate-Bindings *Failure Variable Values-List*

Returns a function of one argument (the culprit) that will bind (via unification) the passed variable to each car of value-set. This function is intended to make writing builtins easier. See, for instance, the definition of Rllib:HMemberP.

UI:Cons-Rhet-Axiom *LHS* &Rest *RHS*

Return a standard Rhet bc-axiom (unasserted) given a list representation, *e.g.*

```
(Let ((var-x (Create-Rvariable "?X")))
  (Cons-Rhet-Axiom (Cons-Rhet-Form 'P var-x)
                   (Cons-Rhet-Form 'Q var-x)
                   (Cons-Rhet-Form 'R var-x)))
[[P ?x] < [Q ?x] [R ?x]]
```

UI:Cons-Rhet-Form *Head* &Rest *Arglist*

Return a standard Rhet form given a list representation, *e.g.*(Cons-Rhet-Form 'P 'A) returns `[P A]`, (Cons-Rhet-Form 'P '(A B)) returns `[P (A B)]`. If the head looks like a builtin, it will be handled appropriately. Rhet variables should be created with Rhet-Terms:Create-Rvariable and EQness will be preserved between calls to this function if the same Rvariable structure is passed.

UI:Grab-Context *Rest-List* &Optional *Default-Context*

Searches the passed list for a :CONTEXT keyword, and returns the context supplied, or the default-context if supplied.

UI:Grab-Key *Key Rest-List* &Optional *Default* &Key *No-Value*

Searches the passed list for a key keyword, and returns the following mark, or the default if supplied. If no-value is non-nil, then if nothing follows the key it is returned

UI:Real-Rhet-Object *Thing*

Returns the Rhet object associated with Thing, *e.g.* the primary element of a canonical. Since primaries are the normal way the Rhet UI deals with function terms, this is a convenient function for the lispfn that wants to print its argument(s). `[Rprint]`, for example, uses this function.

UI:UI-Indexify *Index &Key Force-String*

If the symbol is non-nil, return an index string built from that symbol, else return the default index string.

UI:Truncate-Keywords *Input-List*

Many functions take multiple arguments, via &Rest, that can cause problems when keyword arguments are also supplied. This function truncates a list at the first top-level keyword. Thus, (Truncate-Keywords '(A B C :FOO D)) is returned as (A B C). Note that the new list is freshly consed to avoid any stack problems with destroying a &Rest argument.

Chapter 5

Dictionary of Global Variables

Rhet-Terms:*Current-Context* This is initially set to the root context. It is the context the Unifier is currently working in, and may be reset by the Reasoner whenever a proof demands it. It is distinct from Rhet-Terms:*Default-Context* since the user may be called to manipulate the KB during a proof.

Rhet-Terms:*Default-Context* This is the context used for commands and functions that take an optional context argument, if such argument is left unspecified. Contrast to Rhet-Terms:*Current-Context*. The initial value is T's package, the default most global context.

Rhet-Terms:*Freeze-Package* When freezing an axiom, what package the frozen symbols are interned in.

Rhet-Terms:*Root-Context* The root of all contexts.

E-Unify:*EQ-Error-Object* This is bound to a condition instance that contains the UNDO objects and other information Rhet will need to back out a recursive equality should a problem be discovered.

Chapter 6

Hooks

In order to extend Rhet in an orderly manner (see for instance the TEMPOS system [Koomen, 1989]), a number of hooks into Rhet's reasoning and instantiation functions are defined.

Rhet-Terms:*Create-Individual-Hooks* After a new function term has been added, each function on this list is called with two arguments, the function term, and it's type. The reason for this is if a user-function needs to know about any instances of a given type in Rhet, *e.g.*TEMPOS uses this hook to notify it of new terms of type *T-TIME.

E-Unify:*Add-EQ-Before-Hooks* User hooks to asserting an equality; we call each function on this list with 3 arguments, the two objects being made EQ (the first is a Canonical Name, the second is a Canonical Name or Function Term, which may or may not yet have a canonical name), the third is the CONTEXT-TYPE structure indicating in which context the equality is to be done in. If the hook returns NIL, the equality will not be allowed to proceed, and the user will not be given the option to force it. Otherwise the equality will be added (unless some other error is encountered).

E-Unify:*Add-EQ-Commit-Hooks* User hooks to committing an equality, called with no arguments. The set of all equalities added (see *Add-EQ-Before-Hooks* are being committed to and cannot be backtracked over. Called only for effect.

E-Unify:*Add-EQ-Undo-Hooks* User hooks to undoing an equality, called with no arguments. The set of all equalities added (see *Add-EQ-Before-Hooks* are being undone. Side effects of the before hooks should be undone. Called only for effect.

E-Unify:*Add-INEQ-Before-Hooks* User hooks to asserting an inequality; call each function with 3 arguments, the two objects being made INEQ (Canonical Name or Function Terms), the third is the CONTEXT-TYPE structure indicating in which context the inequality is to be done in. If the hook returns NIL, the inequality will not be allowed to proceed, and the user will not be given the option to force it. Otherwise the inequality will be added (unless some other error is encountered).

E-Unify:*Not-EQ-Hooks* User hooks to proving an inequality; call each function with 3 arguments, the two objects being tested for INEQ (either Function Terms or Canonical Names), the third is the CONTEXT-TYPE structure indicating in which context the inequality is being proved in. If the hook returns NON-NIL, the inequality is considered to be proved, and the result is passed back as the justification.

Note also that the function Typea:Define-REP-Relation allows the user to specify a hook function to be called when an instance is defined over which the relations hold.

Chapter 7

Representations

This section documents the more important internal representations.

In order to provide a more object-oriented interface to the programmer, the representations are somewhat hierarchical, and based on CLOS classes. For example, all structures that are assertable include the class Rhet-Terms:Rhet-Assertable-Term, which include a context slot, a type slot and some other support slots.

Type contains an Rhet-Terms:Itype-Struct structure (*q.v.* section 7.5.1) which indicates the type of the term. It is assumed to be of the T-U type[1], which is the most general type, and the only type a Fact can take on. The Typea:Itype, Typea:Dtype, and Typea:Utype commands set this field for function-terms. It is ignored if the function-term has a canonical name, since the type can change depending on the context (via equality). Since it is of no real use, but the code does use it for historical reasons (the parser creates a fn-term structure with a type, and THEN it gives it a cname), it will go away in some currently unspecified future cleanup.

Context The context this term is asserted to. It is considered valid in all subcontexts, unless it has been explicitly deleted or modified.

7.1 Is *That* a Fact?

Facts are the basic entities for asserted constants in the system (*e.g.* the user asserts that `[F A]`. Strictly speaking, the former is a fact, in the PROLOG sense, while the latter is an expression. The needs of the two entities overlap to such an extent that they share a supertype: Basic-Term. A Basic-Term has the following slots:

Head which is typically the leftmost symbol in its horn clause representation, *e.g.* in `[F A B]` 'F' is the head. It is stored as an atom, though it is usually the printname that is important.

[1] Actually, the constant Rhet-Terms *T-U-Itype-Struct* is used, the most general type.

Args which is a list of the function-terms for each of the arguments to a fact or function term. In the above example, the function-term structures for 'A' and 'B' (in that order) would appear on this list. Canonical names may also appear (*q.v.* section 13.3). Lisp objects may appear only if the Basic-Term is a Fact; Function Terms cannot have non-Rhet obejcts in their arglist[2].

Note that Basic-Terms are a Rhet-Assertable-Thing.

Fact structures inherit from Basic-Term and consist of the following additional elements:

Truth-Value carries the value of :True or :False (*e.g.* when `[NOT A]` has been asserted, the value for 'A' would be :False. The truth-value can also take on the values :Unknown and :Unbound; neither set except internally to Rhet-Terms. :Unknown is a useful default, since fact structures are consed before they are asserted, and :unbound is used to shadow out of a context some parent fact with the same content. That is, since contexts inherit, we track different truth values by using the same fact in each context for a particular fact. We can then look for the fact in the closest context (this context, then it's parent, then it's grandparent) and the first one found is the one we use. To assert a fact is true in some context and retract it in a child turns into putting an :Unbound instance in the child. That way, the parent, and contexts below the parent and above the child, still "sees" the asserted fact, while the child and it's children inherit the unbound one, and treat it as deleted.

Index is a string representation of the index field of a horn clause, prepended with "INDEX-", thus for the horn clause `[[A ?x] <5 [B ?x]]` the index would be "INDEX-<5". The "INDEX-" prefix is used to distinguish it from the heads of facts, both end up being interned in the context the fact is interned in and their values are lists of fact with that index or head, respectively. This is used to make lookup of facts by index or head (the typical case) faster than searching the entire context. In future, it is likely that the number of fields a fact is indexed on will grow. Using the context space to store these keys is a matter of convenience, it would be more general, but more wasteful of resources to cons up a separate hash table for each. Instead we lay contexts on top of the package system (purely to make debugging easier as the UI to packages is more straightforward than to hashtables) and rely on naming schemes to keep us from collisions. Fact heads are any atom, but extremely unlikely to overlap with the indexes, and so long as no one goes and starts naming their facts beginning with "INDEX-¡" that should be safe too. It is likely that eventually a split will be made, since debugging contexts will be rare enough that not having the nice UI will be less of a handicap than these silly rules.

Defaultp is set non-nil if the fact is to be considered *default* (*q.v.* section 13.2.1).

Tag used by TMS.

[2]The reason for this is the main reason for having special Rhet objects in the first place instead of lists: we have to associate properties with terms that are not "atoms" in the lisp sense.

Support used by TMS, this is a list of support structures.

EQ-terms have the following slots in addition to those they inherit from Basic-Term:

Canonical-Name is the canonical-name for this fact (*q.v.* section 13.3), if it has one. Actually, this will be an alist of contexts and canonical names, because a fact may have different canonical names in different contexts.

Distinguished-Type This field takes a type similar to the type field, but is the type specified by the Typea:Dtype command. It is used solely for determining inequality.

In particular, the FN-term is an EQ-term which we actually do equality on at the top-level. It is an EQ term with a flags field that is operated on internally.

7.2 A canonical by any other name...

Canonical names are in some sense the "real" names for objects that have them, and are used by the equality system. Unlike facts, software outside of the Rhet-Terms or E-Unify packages should not be constructing or manipulating these structures directly.

The idea behind them is that two different facts with the same canonical name are equivalent in a weak non-logical sense. That is, given an assertion along the lines of `[EQ [B A] [C]]` then looking up the Fn-term for `[C]` and getting the canonical name, we would get the same canonical name as if we had done the operation on `[B A]`.

There is software in the E-Unify and Rhet-Terms packages for doing anything the higher layers of software may need on the canonical name structure (and they should be used), but for the record:

The Canonical class is an EQ-Term and has the following slots:

Cset This is a list of all the function terms in this union.

Type This is the type of every item in the above set. Note that two facts of different types cannot be assigned to the same canonical name, unless one is a subtype of the other, in which case the fact that is of the supertype is considered to be specialized to the subtype.

Primary This is a Rhet-Terms:FN-Term that is the preeminent member of the set. In the case of, for example having `[EQ [MOTHER-OF TOM] [MOTHER-OF MARY]]` and `[EQ [MOTHER-OF TOM] SUSAN]` SUSAN would be considered the primary. A primary usually has an empty arglist. The primary is not semantically different from any other member of the set, but it is purely a UI issue: it is usually the "prettiest" member of the set to print.

References This is a list of facts that this canonical class references. In the above example, the canonical class of TOM would reference the canonical class of SUSAN, since TOM is an argument of a function term that is in the class SUSAN, and thus if we were to assert that TOM was equal to something else, say THOMAS, we would want (indirectly) `[EQ [MOTHER-OF THOMAS] SUSAN]`.

Unequal This is a hash table used for testing inequality. Basically, if two facts are in the same context, then if they are unequal they will mention each other in their unequal fields. The exception is if they are unequal via Dtype rather than explicit assertion: see the description of inequality markers, below.

Constructor-Set To handle REP style interactions more cleanly, if there is a constructor function associated with the class, mention it here. This can then be invoked, if necessary, to build the actual type. Many times we may be able to do without, e.g. simplifying `[r-human-name [c-human alfred]]` to `[alfred]` w/o constructing anything. We set this to a Form if it exists.

Set-Instance if a set is in this canonical class, it is present here.

Inequality-Markers This is used for speeding up inequality checks. Basically, since a large number of different facts may be in a canonical class, we collect the individual dtypes for the facts in this field. Then seeing if two canonical names are distinguished just involves doing an intersection on the two instances of this field, and if there is something in the resultant set, they are distinguished.

NM-Constraints Constraints that cannot be folded that have been added by Typea:Define-Subtype or Typea:Define-Functional-Subtype are kept on this list, if they are not monotonic. These are checked anytime the canoncial name is changed or updated.

M-Constraints Constraints that cannot be folded that have been added by Define-Subtype or Define-Functional-Subtype are kept on this list, if they are monotonic. These are checked anytime the canoncial name is changed or updated. Monotonic constraints (those that once proved satified will remain satisfied) are removed from this list as they are proved.

7.3 Do not fold spindle or mutilate...

While Facts are restricted to those horn clauses that do not contain variables, most horn clauses will. These sorts of clauses are defined by the Rhet-Terms:Form structure in Rhet. Normally they are arguments to the Unifier, or to the Reasoner (as in (PROVE [B ?x]) or (UNIFY [?x (G . ?y)] [A (G C D)]). Forms are distinguished from lists, in that Forms are what make up clauses in axioms, and can therefore represent calls to builtins or lispfns. Only Forms may be unified against Facts. Lists are basically purely a Lisp type, though they can

be used for dealing with matching several arguments within a form (as a rvariable), *i.e.* via the &rest syntax.

The main reason forms and lists are distinct is that we must distinguish between `[A B C]` which is either the predicate A applied to arguments B and C or the *expression* `[A B C]` which is a unit for equality, vs. the Lisp list, (A B C). The difference is that in the former two, the arguments are essentially decomposable, but in the latter they are not. Where this makes a difference is the difference between `[A (B C)]` and `[A [B C]]`; the former must be a predicate (no equality possible), and has 1 argument, a list, while the latter may be an expression equal to `[A D]` if we know `[B C]` and `[D]` are equal, or even `[F]` if we know `[A D]` is equal to `[F]`. We can attach canonical names to fact structures, but not to lists. So we would run into a problem if `[A . ?x]` were legal (it isn't) and matched with `[A [B C] [D]]`; would ?x be bound to `[[B C] [D]]`? If so, we have the problem that `[[B C] [D]]` may inadvertently get treated as a unit, which it wasn't designed to be: we would have problems distinguishing between `[A [B C] [D]]` and `[A [[B C] [D]]]`; in fact the semantics of `[A]` vs. `[[A]]` vs. `[[[A]]]` become unclear. Further, leaving everything in lisp list syntax gives problems with things like (?x . ?y*(foo bar)) since that looks like something illegal to Lisp. Distinguishing them makes certain things more clumsy (i.e. vararg type predicates), but increases expressivity.

Rhet-Terms:Forms have the following slots:

Value the elements of the form, as a list. *E.g.* from the horn clause `[?x B ?y]` it would be (?x B ?y).

Rvariables A list of the rvariable structures used in the form, directly or in subforms.

Truth-Value As for a Rhet-Terms:Fact, in a goal or clause in an axiom, distinguishes between, say, `[A B]` and `[NOT A B]`

Type an Rhet-Terms:Itype-Struct structure for the type a form returns, if it is known. A Rhet-Terms:*T-U-Itype-Struct* value (the default) means it is not known, and so most general.

Where-Posted If this form is a constraint, then this is the continuation point at which it was posted. This is used to GC constraints off of variables when we want to unbind them.

Forms are probably the most useful of the various structures, as they will be the primary datum being thrown around above the lowest levels. One definition I will add here: The complexity of the form is defined as the lowest level a rvariable is present on. Thus a form with no rvariables is equivalent to a fact in some sense, though it may not be added to Rhet-Terms's KB. A form whose deepest rvariable is on level 1 is called 1-complex, which is what several functions expect at worst case as arguments. Thus if you, as a middle or high level routine have a 3-complex form you want to simplify, and the simplification function only accepts 1-complex forms, you need to extract the deepest part of the horn clause that contains a rvariable (which will itself be 1-complex) and turn that into a form and hand

that to the function. If it simplifies to something without a rvariable, you can use that as a replacement for the form you extracted and continue[3]. Better coding practice would define a recursive version of the function that takes n-complex forms but this may not be practical in some cases. (*E.g.* where anything larger than a 1-complex form is guaranteed to fail).

7.4 Rvariable: Rochester's Weather

One of the entities that can exist on a form is a *rvariable*. What we represent in a horn clause as ?x has a much more complex internal representation. It is (again) a structure[4]. Since we have to distinguish between the printed version of a rvariable, *e.g.* the ?x we use in a query and an axiom we have in our database that uses a rvariable with the same printed representation.

Here are the slots a rvariable has:

Pretty-Name what we see when we print the rvariable, *e.g.* ?x. This does not include what the user interface might want to print out in terms of the type of the rvariable, or the constraints.

Type the type a rvariable has, if it is typed. By default is is of type ***T-U-Itype-Struct*** which is a constant for the universal type T-U. A rvariable may only have a constant assigned to it that is a subtype of or equal to the rvariable's type.

Binding What the rvariable is bound to, if it is.

Bound-P Non-nil if bound (binding could be nil).

Where-Bound What part of the stack we must unwind to to undo binding. This is likely to become a pointer to the proof-tree we were bound at in release 16 since it improves efficiency.

Binding-Form Points back to the fact or axiom used to make this binding.

constraints for the post constraint mechanism, these are the constraints placed on a rvariable. Right now, the constraints themselves have slots indicating the continuation level they were POSTed on.

---

[3] Actually it is highly unlikely that a form with a rvariable would simplify in this sense, but take this as illustrative.

[4] Structures are considered efficient ways to represent things in CL and are implementation dependent representations to keep them that way. That is, on the Symbolics™, it is stored as an vector, but need not be on a SUN™ using Allegro™ Common Lisp.

7.5 It's Not My Type

7.5.1 The Itype-Struct

The Rhet-Terms:Itype-Struct structure is used in most of the above structures to describe the type of the object. Normally it would not be manipulated outside of the TYPEA or TYPE packages. Currently it consists of the following:

Intersect-Types A list of all simple types that intersect to make this type.

Minus-Types A list for subtracting from the above result using a calculus of types. (*E.g.* the type `[foo - bar]` would have type `foo` as a intersect-type and `bar` as a minus-type).

Fixed-Flag Thought to be useful if set that the type is the immediate type of the element, and cannot be further constrained. Thus if I said that Males and Females are an exclusive partition of Humans, (thus anything that is a Human must be either Male or Female) this would let me create an element that is a Human that could not be specialized by the system into either Male or Female[5].

Itype-Structures exist because simple types are precomputed, and we need some way to represent types that are not in the table. In particular, unnamed intersections, and eventually, equations in the type calculus. Thus, for generality, all structures that have the type marked use an Itype-Structure, which the type subsystem can figure out how to deal with in terms of accessing the type table.

7.5.2 The REP-Struct

The Rhet-Terms:REP-Struct is used to maintain the role, constraint, and other definitions pertaining to structured type objects. It has the following slots:

Roles A list of the roles defined on this type. It does not include the inherited roles from parent types, but only the local additions and redefinitions.

F-NM-Constraints This and the following constraint slots hold a list of things to be asserted or proved at various times. This list contains foldable non-monotonic constraints, *i.e.* those for which something can be asserted at instance creation time to make the constraint hold, but must be tested after manipulation of an object, *e.g.* after processing an equality addition.

[5]This isn't actually used except by the Itype declaration, and currently isn't used for anything; it may turn out to be useful, but it hasn't been investigated. One possibility is if you create prototypical things and you don't want the system making these prototypes equal (*i.e.* using equality) to some 'actual' thing. Instead you would get an error.

F-M-Constraints These constraints are foldable and monotonic, that is, they can be asserted at instance creation time, but never need to be tested for. Equality is an example, since equality cannot be retracted.

Initializations A special case of F-M-Constraints that appear as a list of assertions to be made, rather than relations that must hold.

NF-NM-Constraints Non-Foldable Non-Monotonic constraints must be proved rather than asserted, and may change during the proof.

NF-M-Constraints Non-Foldable Monotonic constraints must also be proved, but once proved will not change so need not be reproved.

Functional-Roles A subset of the Roles slot, these roles will have functions associated with them predeclared.

Type The typename itself for this structure. A symbol, usually in the Type-KB package; it's value would then be an Rhet-Terms:Rtype structure.

Relations-Alist An alist of relation names and the list of the relaions.

7.6 Truth

There are *axiom* structures for both forward chaining and backward chaining axioms. These are, for the most part, identical, and the :Include option on defstruct makes the RAX:Basic-Axiom structure shared between the two. The RAX:BC-Axiom structure is just an alias for this, adding two fields:

Cache BC axioms (will) cache their proofs: car is context, cdr is list of proof-cache structures.

References For (future) cache flushing.

while the RAX:FC-Axiom adds a field: RAX:Trigger which is a Form.

Without further ado, the RAX:Basic-Axiom structure:

LHS A Form that represents the left hand side of a horn clause, that is, the conclusion.

RHS A list of forms that are the prerequisites for concluding the LHS.

Index An ascii string, the index field of the axiom.

Context The context this axiom is asserted to. It is considered valid in all subcontexts[6].

[6] Unlike facts, deleting or modifying axioms in a subcontext is NOT supported.

Defaultp If non-nil, the axiom is DEFAULT. It will only be used by the reasoner when default reasoning is enabled.

Global-Vars The rvariables used in this axiom that are global (not local).

Local-Vars The rvariables used in this axiom that are local (*i.e.* in LHS if BC, Trigger if FC, unless inside of a structure (may still be global)) typically this is determined at call-time, for BC axioms, and at parse/compile time for FC.

Key For indexing (hashing), this is the computed hash-key for this axiom.

Axioms are distinguished from facts in that they can contain assertions with rvariables in them, and use any form to specify an axiom. Thus lisp-lists, lisp atoms, builtins, lisp-functions are all legal constituents, except the predicate position must be a builtin, lisp-function or predicate.

7.6.1 Rhet-Set

This is the internal representation for sets in Rhet (a type-updateable term)

RSet A list of objects in the set.

Set-Type Either :ORDERED or :UNORDERED (the default), this indicates if the order of objects in the Set slot is important.

Type A subset of Rhet-Terms:*T-Set-Itype-Struct*. Note that if it is a subtype of *T-Orthodox-Set, then the set can be used in equalities.

Cardinality NIL, if the cardinality of the set is unknown or not fixed. An integer if the cardinality is both known and fixed. Thus, if there are three elements in the Set slot, if Cardinality is NIL, then we don't know how many members are actually in the set (though there must be at least three). If Cardinality is three, then the set is completely known and fixed, and if the cardinality is greater than three, then we know how many members of the set there are (and it is fixed) but we only know three elements so far. Obviously a Cardinality less than the current number of elements in the Set slot is an error.

7.7 Minor Structures

7.7.1 Undo

The Rhet-Terms:Undo structure is used to help undo equalities that are added that cause illegal indirect unions. It records the canonical name of the generated (new) class, as well as the canonical names of both the old classes that were unioned together. Additionally, a 'timestamp' (actually an integer count) is provided as a quick sanity check on the linked list the Undo structure is part of.

7.7.2 Rtype

This structure is used within the Type system. Type "names", *e.g.* the contents of the plus-types slot in an Rhet-Terms:Itype-Struct structure are atoms that are set to an Rhet-Terms:Rtype structure. It consists of the following:

Supersets the defined (immediate) superset of the type, *e.g.* when I create a type, I must state what it is a subtype of. This is assumed to be an immediate supertype.

Subsets A list of all defined immediate subsets of the type.

Index the array index into the type table for this type.

Partitioners A list of the partitioners of this type.

Partitionee If this type partitions some other type, this is that type.

Once again, the above is described for completeness, only the Rhet-Terms and Typea packages are ever expected to manipulate these structures.

7.7.3 Defined Types

The following CL:Deftypes are supported:

Atomic-FN-Term A function term that Satisfies Rhet-Terms:Atomic-FN-Term-P, that is, it is a function term with a null arglist.

Arbform An arbitrary form. This is Deftyped to (OR Form Keyword List Itype-Struct Rvariable). That is, the parameter of this type is allowed to be a rvariable, a fact, a form, a keyword (the way Lisp atoms can be used during unification) or (in the case of arglist processing, or more generally) a list[7]. This is the most general type of object in Rhet for things the user can have in a horn-clause. Note that an Axiom (which is itself a type) is *not* an Arbform. Additionally to support the [Type?] builtin, an Arbform may be an Itype-Struct structure.

Culprit-Type The type of a Culprit, as will be passed to a generator upon needing an additional value. The name is taken from the intelligent backtracking literature...

Context-Type The type of a CONTEXT parameter.

Frozen-Axiom The type of a frozen axiom. Axioms may be frozen (internally) via functions such as RAX:Freeze-Axiom and 'thawed' via functions like RAX:Thaw-Axiom.

Frozen-Form-Part The type of a frozen Form. See Frozen-Axiom, above.

[7] The astute reader will know that Form or Rvariable, being a structure, is also an Atom on the Symbolics and Explorer. The definition is to be taken more literally — as documentation

Generator The type of a generator. Generators are what builtins (among other things) return. They are basically closures. The idea is that one is associated with a subproof, and when called return the next subproof.

Hash-Index What we use to look up facts fast, that is the type of the KB index, not the printed index of a fact.

Legal-Goal The type of an object that Rhet considers to be legal as a goal, as for backward chaining.

7.8 Handling Errors

Errors in the Symbolics and Explorer systems are (currently) built on Flavors[8]. The basic idea is that a condition is signalled which will invoke methods on a flavor appropriately. Depending on the flavor method for the particular error, additional information or a documentation string is provided. The signal, then may be caught by an enclosing form, and possibly handled, *e.g.* if it is proceedable, or possibly the debugger will be expected to deal with it. The base flavor for Rhet errors is Rhet-Terms:Rhet-Condition. Other subflavors (errors) defined on top of this base are:

E-Unify:Rhet-Equality-Problem This is used to signal a problem with recursive equalities. It supplies two proceed options: :undo, which will cause the offending equality to be undone and tossed, and :continue, which will cause the equality to be added anyway, generating an inconsistency.

[8] We expect to begin using the Pitman condition system and CLOS once our development environment supports it, *e.g.* version 18.

Chapter 8

The Reasoner

This package determines the strategy for the proof of a particular goal or subgoal, picks axioms and facts as appropriate for the proof, and implements the horn clause semantics for many functions. (Some horn semantics are wired into the compiled axioms!). It uses the unification subsystem to bind arguments, *etc.* as necessary for interpreted proofs. Like most PROLOG compilers [Warren, 1977a] [Warren, 1977b] [Kahn and Carlsson, 1984] the Rhet compilers generate code with specialized unification code, rather than using the more generalized unifier provided by the Unification Subsystem[1].

The Reasoner will call Lisp functions as declared, rather than attempting a proof using the usual resolution mechanisms. Note that the ability to POSTpone evaluation of axioms (that is, to give a rvariable constraints) is implemented in this package.

Note that if default reasoning is enabled, the Reasoner will return proofs as before, but it may also have used some default rules in the proof - see Reasoner:*Proof-Defaults-Used*. No consistency check is made on the defaults used in a particular proof.

The Reasoner will also handle sets of instances, such as (SET A B C) as a whole to unify with, or to set rvariables to. Sets cannot be asserted, but sets made up only of equality terms (orthodox sets) can be asserted to be equal to another orthodox set or function term.

8.1 Its Flags and Functions

Only variables and functions that are not otherwise described in [Miller, 1990] or the Dictionary of Global Variables chapter are mentioned here.

Reasoner:*FC-Active* Non-Nil When FC is active, so builtins can tell, basically.

Reasoner:*Forward-Trace* This rvariable is set to the list of all assertions made via the Reasoner:Chain function. It must be reset by the user interface to clear it.

[1] Which the interpreter, as well as builtins, in fact, use.

Reasoner:*Possible-Axioms* This is used as an interface between Reasoner:Generate-BC-Proof and [Cut]. It is bound to the alternate axioms at the current level; Cut will clear out other possibilities.

Reasoner:*Reasoner-Disable-Equality* If non-nil, the Reasoner will refuse to use the equality subsystem in order to solve a proof. This gives the effect of a more pure PROLOG like language, with typed rvariables.

Reasoner:*Disable-Goal-Caching* If non-nil, RHET will not cache proved goals and subgoals. Normally this would only be used to debug the caching software.

Reasoner:*Reasoner-Disable-Typechecking* If this is non-nil, all type information is ignored during the proof process. Typed rvariables are still syntactically allowed, but treated as untyped.

Reasoner:*Reasoner-Enable-Default-Reasoning* If non-nil, the Reasoner will use default axioms or facts.

The exported functions are as follows:

Reasoner:Prove-Simple-B *Arbitrary-Form* &Key *Context*
Returns Nil if no proof is found via backward chaining from the goal arbitrary-form, using horn clause semantics. Otherwise, it returns the arbitrary-form (with rvariables bound, and with constraints, if any)

Reasoner:Prove-Simple-All-B *Arbitrary-Form* &Key *Context*
As Prove-Simple-B, but returns all distinct proofs that result in different rvariable bindings.

Reasoner:Prove-Default-B *Arbitrary-Form* &Key *Context*
Exactly like Prove-Complete-B, except that the proof/disproof only occurs at the topmost level, rather than in the entire proof tree.

Reasoner:Prove-Default-All-B *Arbitrary-Form* &Key *Context*
Exactly like Prove-Complete-All-B, except that the proof/disproof only occurs at the topmost level, rather than in the entire proof tree.

Reasoner:Prove-Complete-B *Arbitrary-Form* &Key *Context*
As Prove-Simple-B, but will also attempt a reverse proof using Not forms. Returns Nil if [NOT arbitrary-form] can be proven, :Unknown if no positive or negative proof succeeds, and :Inconsistent if form can be both proven and disproven. Note that while this may seem extremely inefficient, in fact goal caching is presumed to make this less of a problem.

Reasoner:Prove-Complete-All-B *Arbitrary-Form* &Key *Context*
As Prove-Complete-B, but returns all distinct proofs that result in different rvariable bindings, if the forward proof succeeds.

The important unexported (internal) functions are as follows:

Reasoner:Enqueue *Axiom Trigger Context Rank Queue*
Enqueues the Axiom and Trigger for context Context on Queue with Rank. This is used in forward chaining.

Reasoner:FC-Process-Queues *NIL*
Processes the FC queues in priority order. That is, if there is an entry on the PURE queue, it is processed, otherwise the IMPURE queue, otherwise the BC queue. Note that successful axioms will CHAIN result, so we must check all the queues each time. They may have changed! And, since we don't want the stack getting too deep, we return if we are not the top-level queue processor.

Reasoner:Global-Var-P *Var Unbound-Globals*
Returns non-nil if Var is a global, in the PROLOG sense.

Reasoner:Invoke-FC-Axiom *Axiom Trigger Context*
Invokes the Axiom that was triggered via Trigger in context Context. The Axiom may or may not be compiled.

Reasoner:Invoke-BC-Axiom *Failure-Continuation Reinvoke-Continuation Axiom Goal Context*
Invokes Axiom using goal as the invoking goal in context Context. Axiom is expected to return a generator. It will invoke the Failure continuation if Goal does not unify with the LHS of the axiom. Note that Rhet generally "pre-unifies" a goal with axioms by it's indexing and axiom selection strategy. Generally this assures that at least the types of the arguments in the goal and LHS of the axiom are appropriate.

Reasoner:Pause-Check *Results*
Tests to see if Reasoner:*Reasoner-Pause-Function* exists, and if so, calls it.

Reasoner:PBM-FC-Link *Failure-Continuation Reinvoke-Continuation Legal-Goal*
This function is the builtin PROVE for FC. It maps into a BC proof, after setting up things right.

Reasoner:Recursive-Interpret-FC-Clauses *Axiom RHS-Clauses-Left Justifications Failure-Continuation*
This is the guts of interpreting clauses. We have to do it this way to get continuations to work right, since we can only continue to things that are active (a pity, too). RHS-Clauses-Left are the uninterpreted clauses of Axiom. We collect proved forms on the Justifications list. If we can't prove the Car of RHS-Clauses-Left, we invoke the Failure-Continuation.

Reasoner:Recursive-Interpret-Bc-Clauses *Failure-Continuation Unbound-Globals-Continuation Left-To-Prove*
Does the gruntwork of BC proofs: proves the next clause on the RHS of the Axiom we are proving, saving continuations, *etc.*, for restarting on internal closure rvariables.

Note that we don't have to return anything, the destructive bindings of axioms will do it all. This function in reality returns a generator.

Reasoner:Uncrunch *LIST*
Takes a justification list and strips out anything that isn't a fact.

8.2 Its Design

The Reasoner implements a somewhat enhanced proof procedure compared to the old HORNE system. That is, given some subgoal, it will first see if it, or something equivalent to it is asserted (unless it is a Lisp function), and if so it provisionally succeeds. If not, it checks to see if the inverse of the goal is asserted (or something equivalent to it), and if so fails. If not, it will attempt to prove the axiom. If it cannot prove it, it will attempt to prove the inverse (if the appropriate function was called from the QUERY interface), and if it still can prove nothing it will return the fact that it can neither prove nor disprove the goal, rather than simply fail.

Note also that the Reasoner may invoke the equality subsystem itself rather than relying on the Unifier to do it, should rvariables not appear in either expression, or for other reasons depending on the wired in heuristics.

The heuristics the Reasoner uses to determine how it goes about attempting to prove or disprove some subgoal should be made easily modifiable, so experience with the Reasoner will allow improvement in these heuristics without a major rewrite. In fact, right now it is concentrated in Reasoner:Generate-BC-Proof, though Reasoner:Prove-Based-on-Mode and the interpreter have something to say about it. (the compiler will probably have more).

FC axioms that use the PROVE form get the first proof via BC. Backtracking of the high-level PROVE form is NOT supported[2]. Further, this form is assumed not to have side effects. Similarly for builtins, *etc.* in the RHS of a FC axiom.

Rvariables are bound directly, and destructively, in the rvariable structure, more information is given in section 3.3.1 [Bruynooghe, 1982] [Mellish, 1982] [Kahn and Carlsson, 1984] [Sterling and Shapiro, 1986]. All internal functions and builtins handle arguments that have this structure, such that intelligent backtracking [Warren, 1986] [Bruynooghe and Pereira, 1984] [Cox, 1984] can be implemented. The idea is that all functions that can backtrack take a failure continuation as an argument. If they cannot find any proof for the form handed them, they invoke the continuation. If they can identify a particular *culprit*, however, a slot on the rvariable contains the continuation that should be invoked.

In order to make all this work, we internally do two things: proving something via FC or BC is normally finding some goal, and getting a closure which is used as a generator to give us successive proofs of the goal (possibly with different global rvariable bindings). For example, when we call RKB-LOOKUP on `[R ?x ?y]` we might get ?x/T ?y/A first,

[2] *Because each prove from FC requires establishing a new context for BC, which cannot be appropriately saved for backtracking*

and on reinvokation ?x/T ?y/C, *etc.*. Similarly, if there were an actual RULE that did this proof, we would find it by the usual mechanisms and get back a closure: the interface to an axiom (compiled) that does BC proofs of some goal is really identical to the interface to RKB-LOOKUP, the only difference is that the latter is more efficient, it just checks the KB. Once a generator has proven (or failed to prove) some goal, we cache it [Fagin, 1984] so we can reuse the result elsewhere in the proof tree as needed[3].

See the library section 10 for a description of the various macros that are defined to manipulate these closures such that the failure continuations work correctly.

8.3 Unimplemented

Compiler currently just compiles a call to the interpreter in.

[3]Well, we want to, it isn't implemented yet.

Chapter 9

The Axiom Database Subsystems

The axiom database keeps the forward chaining and backward chaining axioms segregated, although they are stored similarly[1]. Note that where triggers, LHSs *etc.* are supplied, these are to be Forms. These are somewhat restricted: typically triggers, and LHSs that are supplied must have their Form-Head bound to a function term, rather than an arbitrary term (*e.g.* a Rvariable, another Form, *etc.*).

Contexts are more restrictive in axioms than they are in facts[2]. The main restriction is that, unlike facts, axioms cannot be deleted in a specific context, nor can they be shadowed with an alternate form[3]. Axioms that are asserted to a context are accessible to all child contexts. To prevent an axiom from being accessible in a particular child, it would have to be removed from the parent.

9.1 Using Them

Here are the flags and functions that are exported as they are currently defined, subject to change, *etc.*

Rhet-Terms:*Frozen-Var-Htable* A hashtable of the rvariables that are frozen for the current form.

---

[1] see section 9.2.

[2] This may be something worth discussing - we didn't see a particular need to support something more, which could, of course, be done, but would be harder. Just to be able to make a particular axiom invisible from some child on down would not be too hard - basically involve adding a 'lower' bound context list to the axiom structure, though it would make lookup more cumbersome. We would not need the complex approach that was taken with facts, since axiom lookup is much less frequent; we can afford to be a little inefficient for the sake of space and code complexity. Second, if needed, the prover is the real mechanism that would need to be fast here, and the hashtables it uses COULD be copied into and updated for specific contexts if necessary.

[3] Again, we thought such things unnecessary. It is even unclear how they will be used for facts: they exist as a side effect of other decisions.

Note: All functions take an optional keyword argument :Context, to specify the context to be used for the command. The default used is the value of Rhet-Terms:*Default-Context*, since these functions may be called from the user interface.

All of the following functions return a list of the axioms they operated on. Those that take a defaultp flag will only operate on axioms marked default if it is set. The removal functions ignore the default flag on the axiom. For example, an axiom added via RAX:Add-Term with Defaultp set to non-Nil, will not be returned by an appropriate RAX:List-All-Axioms-B even if it matchs, if the List function is called with Defaultp Nil. But a remove function that matches will remove this axiom, since remove functions ignore the Default flag.

RAX:Add-Term *Basic-Axiom* &Optional *Context* &Key *Justify*
Adds the axiom (forward or backward) to the appropriate database, after doing consistency checking on it. This usually involves compiling the axiom. Implementation note: Note that this function need not be called to add an already compiled axiom, such could be "dropped" directly into the KB when the compiled file was loaded. Thus, care must be taken in the axiom compiler to appropriately wire in the compiled form into any structures. Returns the new axiom.

RAX:Remove-Axiom-F *Trigger* &Key *Context*
All axioms that match the literal trigger and are in the passed/default context are removed from the database. Returns a list of the axioms.

RAX:Remove-Axiom-B *LHS* &Key *Context*
All axioms that match the literal form (on the left hand side of a horn clause) and are in the passed/default context are removed from the database. Returns the list of the axioms so removed.

RAX:List-Axioms-F *Trigger* &Key *Context Defaultp*
The axioms that match the literal trigger are returned. Unless Defaultp is specified to be T, axioms added with Defaultp NIL are NOT returned.

RAX:List-Axioms-B *Form* &Key *Context Defaultp*
The axioms that match the literal form are returned. Unless Defaultp is specified to be T, axioms added with Defaultp NIL are NOT returned.

RAX:List-All-Axioms-F *Atom* &Key *Context Defaultp*
All axioms with atom as the head of a trigger are returned. Defaultp works as for List-Axioms-F.

RAX:List-All-Axioms-B *Atom* &Key *Context Defaultp*
All axioms with atom as the head of their left hand side are returned. Defaultp works as for List-Axioms-B.

RAX:Remove-All-Axioms-F &Key *Context*
All axioms directly in the passed/default context are removed from the forward chaining KB, whether added via Add-Axiom or directly by loading a compiled file. Returns no value.

RAX:Remove-All-Axioms-B &Key *Context*
All axioms directly present in the passed/default context are removed from the Backward chaining KB, whether added via Add-Axiom or directly by loading a compiled file. Returns no value.

RAX:List-Axioms-By-Index-F *Index* &Key *Context*
All axioms with matching index are returned.

RAX:List-Axioms-By-Index-B *Index* &Key *Context*
All axioms with matching index are returned.

RAX:Remove-Axioms-By-Index-F *Index* &Key *Context*
All axioms with matching index and in the passed/default context are removed from the forward chaining KB. A list of the axioms is returned.

RAX:Remove-Axioms-By-Index-B *Index* &Key *Context*
All axioms with matching index and that are in the passed/default context are removed from the backward chaining KB. A list of the axioms is returned.

RAX:Compile-Axiom *Basic-Axiom* &Optional *Context*
Called internally by Add-Term, it is provided so the user interface can provide a function compiler and dump forms as necessary to a BIN file. Implementation note: part of compiled form may be an (eval-when :load) so when the form is loaded it will wire itself appropriately into the KB. The user interface is responsible for Remove-Axiom-Xing the old definition of a function that is changed by the user.

RAX:Generate-Key-From-Term *Term*
This function takes a goal and generates the appropriate key to use in finding appropriate axioms to use in it's proof or chaining. The head of the goal is not used in creating the key except indirectly, as this is normally passed as a separate parameter to RAX:Get-FC-Axioms-By-Index anyway.

RAX:Get-FC-Axioms-By-Index *Head Key Context*
The Index referred to by the name of this function is the hash index, not the axiom index. This is the Key parameter, and is calculated based on the trigger we are attempting to find an axiom to chain on. Given the head and a good key, we try to pick only those FC axioms that are likely to fire on the trigger. We do this by doing stuff at compile-time, *e.g.* if the trigger is `[F [S ?x]]` we know that the argument to the head F must be a structure and we index our axiom in the axiom KB based on that. During a proof, if we are given an actual trigger of `[F :A]` we will calculate the head to be `[F]` and the key is an atom, which is not a structure, and thus we will not pick this clause as an appropriate one to fire.

RAX:Get-BC-Axioms-By-Index *Goal Context*

The Index referred to by the name of this functior is the hash index, not the axiom index. This is calculated based on the goal we are attempting to find an axiom to prove. Given the head and a good key, we try to pick only those BC axioms that are likely to prove the goal. We do this by doing stuff at compile-time, *e.g.* if the goal is `[F [S ?x]]` we know that the argument to the head F must be a structure and we index our axiom in the axiom KB based on that. During a proof, if we are given an actual goal of `[F :A]` we will calculate the head to be `[F]` and the key is an atom, which is not a structure, and thus we will not pick this clause as an appropriate one to use.

The following functions are not exported. They should not be used, and are documented here for completeness and allow possible code-sharing in the future (if you know what is already written, you won't need to rewrite it!).

RAX:Ill-Formed-FC-Axiom-P *FC-Axiom*

This does some rationality checks on the FC-Axiom structure. Returns T if there is a problem

RAX:Ill-Formed-BC-Axiom-P *BC-Axiom*

This does some rationality checks on the BC-Axiom structure. Returns T if there is a problem.

RAX:Index-Axiom *Index Axiom KB-Name Context*

This function indexes the axiom into the KB-Name using the passed Index. It is used by both the FC and BC front ends, since they are indexed identically.

RAX:Trim-Unaccessible-Axioms *AA-List Context Defaultp*

Given a list of axioms, this returns the list with unaccessible axioms deleted; that is, axioms that would not be accessible to the passed Context, or are marked as default when Defaultp is Nil.

RAX:Find-BC-Axiom-Locals *BC-Axiom*

Given a BC axiom, figure out what the local rvariables are, and return them. Note that all we might think we have to do is return the rvariable list for the LHS, but that isn't sufficient: a rvariable could be for passing info upward on the LHS, which is effectively a global. That is, it will only be bound to a rvariable on invocation. Assume the compiler does this; worst case individual axioms have to do their best.

RAX:Find-BC-Axiom-Globals *BC-Axiom Locals*

Given a BC axiom and the local rvariables therein, compute the globals used.

9.2 Figuring Out How They Do It

Implementation note: Contexts are not handled as nicely as with facts, because of the potential problems with ordering axioms in multiple namespaces. Relevant context is kept in

the RAX:Basic-Axiom structure, and part of the axiom's program checks it's own accessibility in the Rhet-Terms:*Current-Context*. If an axiom is NOT accessible, it fails quietly. (That is, it does not signal an error). Also, the Reasoner will not in general attempt to invoke an axiom that is not accessible, so this may not be strictly necessary.

Another implication of this handling of contexts is that, as described above, we only handle the case of an axiom being interned in some context and accessible to all children of that context (checked via Rhet-Terms:Accessible-Context-P). This kept the code very simple, and lookup reasonable. Something more complex is possible, at a concomitant cost in code complexity and time for usage.

All of the functions in RAX are quite straightforward. The basic plan is to store the axioms, compiled, in a package (either FC-Axiom-KB or BC-Axiom-KB) and invert them based on their index and either their LHS (for a BC axiom) or trigger (for a FC axiom) which is how the reasoner will typically want to look them up. Many functions operate on all of the symbols that have been interned in these packages (such as delete-all). They access the symbols one at a time and see if they meet the proper criteria. The heads and indexes are interned directly; their value is a list of the axioms they match.

9.3 Future Work

Right now no mechanism is provided for changing the axiom order[4]. We plan on leaving this undefined to allow a future parallel implementation.

The Ill-formed axiom functions currently do nothing (always fail) pending future need, when the user interface will call some functions directly, or need something to check the user's horn clause for legality.

The axiom compiler only compiles a call to the interpreter on the passed axiom.

[4]clearly this is only an issue for BC axioms, as all FC axioms whose triggers are matched will be fired, although the order of firing might be an issue to TMS.

Chapter 10

The Language Definition Library

10.1 Using the Library

Section 3.4 describes the macros and functions provided to support builtins.

The above, as well as other code, also use the following special vars:

Rllib:*Debug-Continuation-Establishment* if non-Nil, it is expected to be a function, that is called with a format string and arguments each time a continuation is established.

Rllib:*Debug-Continuation-Compile-Flag* Separates whether or not we compile in the debug code, and if it's active.

There are also functions associated with each of the builtins described in the User's manual. Since these simply do what was indicated there, documentation is not repeated for them here. Note that these lispfunctions that are defined as builtins need not have the same name as appears to the user. For example, Hequalp is the lisp function that handles the builtin EQ?, while the lisp function name and the Rhet builtin name for Bagof are identical. Each take two additional first arguments before their documented arguments in [Miller, 1990]: their first argument is a failure continuation, which is invoked when the builtin cannot successfully prove something, and a reinvokation continuation, which is passed as information to the builtin: it is the continuation that will be invoked to request an additional proof from the builtin. All of the builtins return a generator that supply successive "proofs" when invoked. Note that Rllib:Define-Builtin declares if a builtin has side effects (and if so, what it's undo function is). All are expected to handle backtracking. (If it does not make sense for a particular builtin to backtrack, it still returns a generator, that will only generate one value then invoke the failure continuation if reinvoked).

10.2 Design Details

For the most part, this code is reasonably straightforward. All functions adhere to the interface of taking a continuations parameter, and returning a success/failure indication, and the closure if needed. A good proportion of the code assumes that by the time the basic functions are called, any rvariables have been expanded (replaced by their bindings), but this is changing to take advantage of intelligent backtracking. Functions which do not bind rvariables, will often not bother with a closure (since there is no backtracking to be done anyway).

Recursive proofs are typically done using the Reasoner:Prove-Based-On-Mode call, though FORALL which does complete BC proofs, uses the higher level functions.

Certain functions expect facts, even if a canonical name might have been used. For example, given that `[A B]` and `[C]` have been asserted to be EQ, which is used for the **Hatomp** test to determine atomicity?

10.3 Left To Do

- The macros need enhanced to do the right thing if there are side effects. This is still an open problem.
- Caching needs to be handled right, particularly if there are side effects; want to know if caches must be flushed (and if so, which ones). This may involve enhancing define-builtin to advise how much of a flush is needed.
- Right now, builtins are not unified against, they are just called with the entire arglist. We may want to match the arglist against the types declared to define-builtin, and complain if there is a problem.

Chapter 11

The E-Unification Subsystem

The job of the Unifier is to take two forms and calculate bindings for each rvariable needed to unify them. Equal rvariables in the two forms are assumed to be the same. Type restrictions of the rvariables are taken into account. With the structure copying approach, the unifier no longer returns the bindings, but rather the rvariables are destructively modified to contain their bindings as part of their structure.

Interfacing to the Rhet-Terms Subsystem, functions in this package will add new equality assertions, or simplify fully grounded expressions for the Reasoner and the Unifier. It also provides equality-based retrieval to the Reasoner and Unifier, that is, sets of facts based on a given canonical name. This may include, for example, possible parameters to a function such that the function will then be equivalent to a given generic name. Implementation note: Note that there are no compiled forms specified for defining equalities. Because most of the work would have to be done on the load of the so-called compiled form, no advantage could be seen in providing it. Instead, the world save[1] is considered an adequate and much faster way to store equalities between sessions.

11.1 Overview

This subsystem provides E-Unification to the Rhet system. E-Unification is further described by Kornfeld [Kornfeld, 1983], and is also taken up, for example, in [Haridi and Sahlin, 1984]. The algorithm Rhet uses to maintain equalities is fairly simple in concept, though more difficult in practice. The basic idea is that two terms that have been asserted to be equal, have the same canonical name. Rhet's algorithm, unlike, say, Union-Find (see [Hopcroft and D., 1979]) does all of it's work on assertion, in order to achieve O(hash) lookup time for equalities. Thus, to check if two terms are equivalent, we merely have to get their canonical names, and see if they are the same. This is made somewhat more

[1] A provision of the lisp machines, and certain other lisps, is the ability to snapshot the entire environment into a file, which can be restored at a later time.

complex since the canonical name may depend on context, however, for some fixed number of contexts lookup is still O(hash)[2].

These canonical names that are blithely compared by unification, are actually somewhat more complex. The canonical name structure contains a list of all the equality terms that hold the canonical name, a primary element of that set (the one the user would be most likely to want to have presented for the set, *e.g.*, given that `[ADD-EQ [Father-of John] Fred]` we'd want the system to present `[Fred]` rather than `[Father-of John]` on output, so it would be made the primary. The canonical name structure also contains a list of equality terms that are referenced by this one. In our `[Father-of John]` example, `[John]` would be one of the references. Rhet has to track references since if `[John]` is asserted to be equal to something else, say `[Sam]`, then `[Father-of Sam]` would also be in `[Fred]`'s equality class. This is particularly important with contexts, since we may have, for instance, already have inconsistently declared in some child context that `[Fred]` and `[Father-of Sam]` are *not* equal! Or, further, `[Father-of Sam]` may already have some independent equality class, that must now be unioned in with `[Fred]`'s canonical class. Naturally, all of this must be tracked by context, so typically, if a union is to be done in a particular context, then all of the child contexts to it are processed first, if any of the canonical names are also aliased in some sense in one of the children. This is complicated by the fact that a canonical set in a particular context may only be a small part of the set in a child context, since further unions may be effective there.

Last, rather than being careful to reuse the canonical classes as the standard Union-Find algorithm is, we always generate new classes, because this makes it easier to undo the union request, if during processing we detect that a recursive union would fail. This could happen either because of a direct inequality assertion, or because of an inconsistency with a predicate. An example of the former was presented above, and an example of the latter might be attempting to assert that `[Sam]` and `[John]` are equal, when `[Owns-Helicopter-P Sam]` has been asserted, as has `[Not [Owns-Helicopter-P John]]`.

11.2 The Usage

Note: All functions that take an optional argument for context, will default to the value of Rhet-Terms:*Current-Context*[3].

The following globals are defined:

E-Unify:*Enable-HEQ-Warnings* If non-nil, warnings will signal an error. Otherwise they are ignored.

---

[2]The time to find the canonical name of a term is proportional to the time to run down an assoc list of contexts and canonical names that is associated with the term. Since a particular term typically has few canonical names, this is a reasonable solution. Should our typical case exceed about 70 or 80 names on the lisp machine, an additional hash table may become beneficial.

[3]Not to be confused with *Default-Context*.

E-Unify:*Trace-HEQ* If non-nil, a diagnostic message is printed whenever a new equality is added.

Here are some of the functions the Unifier package provides (that aren't documented elsewhere).

Note: Many functions take an optional keyword argument :Context, to specify the context to be used for the command. The default used is the value of Rhet-Terms:*Default-Context* for the equality definition functions, and Rhet-Terms:*Current-Context* for the equality test and lookup functions (normally called from the reasoner).

E-Unify:Define-Equality *EQ-Term1 EQ-Term2 Context* &Key *Handle-Errors*
Uses E-Unify:Def-EQ to assert in the current (or specified) Context, that two terms are equivalent, and of the appropriate implicit type. Note that if the type of the two simple EQ-Terms are not compatible, an error is signaled. The function returns the canonical name of the the new class. Note: should one of the arguments already be in a class, the other is added to it. Should both be in different classes, they are collapsed (via Rhet-Terms:HN-Union), and the surviving class name is returned. If Handle-Errors is supplied and non-NIL, then a condition is built to handle any problems adding the (recursive) equality. Otherwise the call is not considered to be at the top-level, and the existing binding of E-Unify:*EQ-Error-Object* is used.

E-Unify:Def-EQ *Canonical-Name Function-Term Context*
This function will call Rhet-Terms:HN-Union on the Canonical-Name and Function-Term after verifying that the type (implicit) of the Function-Term is consistent with the Canonical-Name. E-Unify:Def-EQ returns the new canonical name assigned to this new class, and will not otherwise return (Resume will unwind to top, Abort leaves everything alone, and Continue will force the union to proceed even with the inconsistency.). Def-EQ does NOT set up a new Rhet-Terms:Undo structure, as does E-Unify:Define-Equality, thus is called from inside the Rhet-Terms package by Rhet-Terms:Union-References. It does, however, fill in details of the UNDO structure and condition object bound by Define-Equality.

E-Unify:Hgenmap *Canonical-Name 1-Complex-Form* &Key *Context Just-One*
Returns a list of the facts in the class Canonical-Name that fit the 1-Complex-Form as a template (including any typed rvariables, appropriately). As a second value, returns the bindings needed to generate each entry in the first list. Thus, it will index into the representation of the function to select rows and columns of argument pairs whose values are equivalent to the Canonical-Name - it is not really unification. If Just-One is specified and non-NIL, then the call is actually only being made to prove there is some member of the class that fits the form (used when establishing constraints).

E-Unify:Hgenmapall *1-Complex-Form* &Key *Context*
As E-Unify:Hgenmap, but it is not constrained to return only forms that are equivalent to some canonical name. Rather, an assoc list of all canonical names as keys (as known

for the head function of the 1-Complex-Form) and a list of lists of possible rvariable bindings that would make the 1-Complex-Form equivalent to it are returned. NOTE: This function will not work as coded if any of the args in the 1-Complex-Form are themselves canonical names.

E-Unify:Test-Equality *Item1 Item2* &Key *Context*
This function returns a boolean indicating if the two equality terms (Item1 and Item2) are compatible. That is, if the two facts are in the same canonical class, the fact is in the passed canonical class, or the two canonical classes are the same in this context.

E-Unify:Check-Compatibility *Can1 Can2* &Optional **(Context *Current-Context*)**
Tests that the two canonical classes are compatible, that is, there is no fact alpha s.t. not (alpha) is a member of the other class. Returns NIL if everything ok, otherwise first instance of problem detected.

E-Unify:Undo-Current-Equality *Context*
Undoes whatever work we've done so far adding an equality.

E-Unify:Commit-Equality *Context*
Garbage Collects the stuff we left lying around in case of an undo. Specifically, process the undo list, and delete references to old canonical names in fact structures.

E-Unify:Delete-Obsolete-Cnames *Fn-Term Cname-Reference*
Take a function term and process it's canonical-name alist: delete obsolete references to Cname-Reference.

E-Unify:Simple-Unify *Type Arbitrary-Form* &Key *Context*
Used when unifying with a simple typed rvariable. The function checks that the Arbitrary-Form is compatible with the Type (an **Itype-Struct**) and returns success indication, then Arbitrary-Form unsimplified as the second value, unless some simplification is necessary to check the type (as in the case of a function defined to return different types depending on the types of it's arguments). The function returns three values, the first is Nil if the type of the Arbitrary-Form is not compatible with Type. Note that the Arbitrary-Form can be a Form, a canonical name, or a fact. The third value returned is the most constrained type of the object and the type passed. Unless the object is a rvariable, intersection at type T-Nil is considered unification failure.

E-Unify:Complex-Unify *Arbitrary-Form1 Arbitrary-Form2* &Key *Context*
This function unifies two arbitrary expressions, using equality to simplify or support as necessary. The function may call itself recursively. The function returns Nil if no unification is possible between the two forms specified as its first value.

E-Unify:Unify-Without-Equality *Arbitrary-Form-1 Arbitrary-Form-2* &Key *Context*
Like Complex-Unify, but refuses to use the equality subsystem. Thus, if the two forms do not directly unify, this function returns Nil[4].

[4]This function is provided so the Reasoner package may support alternative proof strategies, and so the user can disable the equality system temporarily if desired.

E-Unify:Generate-Bindings-Alist &Rest *List-of-Forms*
Given one or more forms, the function returns an Alist of the rvariables mentioned and their bindings.

E-Unify:Unify-Arbform *Arg1 Arg2* &Optional *Context*
Attempts to unify two arglists of two different forms. Return value is a boolean indicating success. Arg1 and Arg2 are lists, Forms Rvariables, facts, or canonical names.

11.3 The Description

E-Unify:Unify-Arbform is a generic function that is somewhat complex. If one of its arguments is a Rvariable, it calls E-Unify:Rvariable-Match on the rvariable and it's other argument. If both arguments are Forms, it calls a particular version of Unify-Arbform on them, and otherwise attempts to simplify any argument that happens to be a Form (via E-Unify:Simplify-Form). If it ends up with two fact or canonical names, it calls E-Unify:Test-Equality on them. Otherwise if we are still dealing with lists of objects, we call ourselves recursively on our Car and Cdr.

E-Unify:Rvariable-Match checks to see if the rvariable is already bound, and if it is, calls Unify-Arbform on the bound value and the item passed. (If it succeeds, we know it was an equivalent object so we are OK). Otherwise we bind the rvariable to the form, providing the types are consistent. To prevent problems we don't bind a rvariable to itself, we just succeed.

E-Unify:Contained-In makes sure we aren't, for example, attempting to unify ?x with [A ?x] which would otherwise recurse forever in some sense. It checks that the rvariable is nowhere present in the form, by recursion on the form if it extends over multiple levels. If it finds a rvariable in the form bound to the rvariable we are checking for, this also counts. This occurrence check is omitted, for efficiency, if E-Unify:*Omit-Occurrence-Check* is non-nil.

E-Unify:Define-Equality basically does some simple argument verification, looks up the canonical name of one of it's arguments (or generates one if neither has them) and calls E-Unify:Def-EQ on the results. The only other thing this function does is clear out the Rhet-Terms:Undo structures from the last call, preparing for a possible failure along the way.

E-Unify:Simplify-Form takes is argument and attempt to translate it into a Fact that Rhet-Terms:Find-Fact can be called on. This may involve calling itself recursively if one of the arguments in the passed Form involves a Form itself. It returns the canonical-name of the fact if it does get translated, and the fact has a canonical name.

E-Unify:Hgenmap and E-Unify:Hgenmapall as they are currently implemented are very straightforward, and probably much slower than they need to be. Hgenmap gets the set of members of the class Canonical-Name, and then invokes the Unifier's E-Unify:Term-Unifies-With-Form-P to check if each member fits the 1-Complex-Form.

E-Unify:Hgenmapall goes and looks up everything in the KB that has a head that unifies with the passed head in the 1-Complex-Form, and then checks to see if the rest of the item unifies with the 1-Complex-Form as well, and returns it on an alist with its canonical name.

E-Unify:Test-Equality is pretty straightforward. It gets the local definitions for the passed Items, and sees if they have the same canonical name, or are (now) Eq.

Chapter 12

The Type Assertion Interface

12.1 Interfacing to the Interface

Only important functions not already described in the user's manual are presented here.

Typea:*Function-Table* Setqed to the hashtable holding function typing information.

Typea:Make-Function-Table
Creates a hashtable and binds it to *Function-Table*.

Typea:Init-TypeA
initialize the type-assertion system.

Typea:Find-Or-Create *name*
Return the Rtype for the named type. Create a new type if necessary.

Typea:Set-RType-Relation *rtype1 rtype2 relation*
Make the two appropriate entries in *TypeKB* for the relation between the two types indicated. Relation is either the name of the intersection, :Subset, :Superset, :Disjoint, :Nondisjoint, or :Equal.

Structured types have several interesting internal functions, though they would not normally be called by the user[1].

Typea:Run-Foldable-Constraints *Constraint-Form Instance*
Takes a foldable constraint (a builtin or lispfn with an assert fn, or a Rhet predicate that just needs asserted) with the variable ?self, binds the variable to the pased value and adds the result.

[1]The advanced user may wish to Advise them.

Typea:Process-Roles-of-Instance *Instance Type Roles Context*

Where Roles is of the form (r1 v1 r2 v2 r3 v3 ...), for each (r v), add `[Add-EQ [F-R Instance] V]`. While we are at it, process constraints and initializations. This is the function that is run when we create an Instance of a Type.

Chapter 13

The Rhet Term Subsystem

13.1 Types

This subsystem is not fully realized. All of the functions are implemented, but they will only work on simple type expressions, rather than a more complete type calculus we hope to eventually support.

13.1.1 Overview

The basic algorithm for the type subsystem is fairly simple. Given two simple types, we look in a precomputed two dimensional table, and find their relationship there. Like the equality subsystem (*q.v.*), the equality system has been optimized for fast lookup, and the work done when type relationships are asserted to precompute the table. Since the type table is built offline (the reasoning system not being monotonic over changes in the type table), this should not be an issue. Type reasoning could be implemented more completely and slowly using the usual horn clause resolution mechanisms, however, the primary idea is to make a limited amount of reasoning about the types of equality terms and rvariables to be quite fast.

Complex type expressions, *e.g.* those involving subtraction, are computed at and reasoned about at run-time. At the moment, the type reasoner is only capable of dealing with type intersection, and type subtraction, and that only in a very simple precedence form. All types ultimately must resolve to basic types (that are stored in the table) and come out to a set of types that are intersected, and a set of types that are subtracted from the intersection to give the result. Thus, *e.g.*, the type (Human Female - Moron) would intersect the types Human, and Female (possibly coming up with Women, if that were appropriately asserted), and subtracting the Moron type from it. But the similar expression (Human - (Female Moron)), that is Humans that aren't Female Morons is not expressible. Instead the system would have to treat it as (Human - Female Moron) which given the usual intuitive definitions would be the same as (Human Male - Moron) which is quite different since we wanted to only eliminate female morons, and instead eliminated all females and all morons!

13.1.2 Interface

Some of the following are exported mainly for the use of the TYPEA package. It is unlikely any of the flags other than Rhet-Terms:*T-Nil-Itype-Struct*, Rhet-Terms:*T-U-Itype-Struct*, and the other predefined Itype structures will be needed by any other packages and should be considered exclusively for the use of TYPE and TYPEA.

Rhet-Terms:*T-Nil-Itype-Struct* An empty type, the least general of all types.

Rhet-Terms:*Last-Index* The index of the last type created. (T-U (0) and T-Nil (1) are already there, as are the Lisp related and other predefined types)

Rhet-Terms:*TypeKB* This is the type array. It supports the following relationships:

:equal for two types that are identical.

:superset if for entry [a b] a is a superset of b.

:subset for the opposite.

:intersection if [a b] may intersect, but the intersection is unnamed.

:partition if for entry [a b] b is a subset of a, and for all other subsets of a there is no overlap (part of a cover of a) note that [b a] will be of type :Subset, and :Partition implies superset.

a non-keyword symbol if [a b] intersect then the intersection has the name of the non-keyword symbol (which is also a type). That is, when we look up the type relationship in the table, if the entry is not a keyword symbol, then it is the type that represents the intersection.

:exclusive if [a b] do not intersect.

:unknown if the relationship is undefined, and could not be derived.

Rhet-Terms:*T-U-Itype-Struct* An instance of the universal type, the most general (rhet) type.

Rhet-Terms:*T-Atom-Itype-Struct* An instance of the type for Lisp atoms (must be in keyword package to be recognized).

Rhet-Terms:*T-List-Itype-Struct* An instance of the type for Lisp lists.

Rhet-Terms:*T-Lisp-Itype-Struct* An instance of the type for any Lisp object.

Rhet-Terms:*T-Anything-Itype-Struct* An instance of the type for any object whatsoever, Lisp or Rhet.

Note that all other predefined types are also present as *T-Typename-Itype-Struct*.

Exported functions of this package are identical (possibly names changed) to the functionality described for the types system in the Rhet User Manual.

The following are non-exported functions and variables related to TYPEs.

Rhet-Terms:*Type-Mode* (variable, initially default) :**Default** to assume type with unknown relationship are disjoint; :**Assumption** to assume they overlap.

Rhet-Terms:Init-TypeKB
Initialize the type-kb system.

13.1.3 Remains to be Done

- Complex types are not completely handled; we probably want to expand to a general type calculus at some point.

13.2 Facts, Function Terms and Other Instances

Using the package system[1], this subsystem's job is to keep facts about axioms and equalities relative to some context. Facts that are added to a particular context should shadow any facts with the same name from a parent context. The request for modification of a structure in a parent context should (logically) cause a copy of that structure to be placed in the current (requested) context's package, and the modification done to this copy (thus keeping the parent's copy intact).

Rhet-Terms hides details of how the KB is stored, for example, if facts are stored in an association list, a hash table or an array.

13.2.1 How to Use It

Facts added with the defaultp argument bound to t are only accessible with the same flag binding on access functions. This prevents the Reasoner from seeing default facts if it is not currently doing default reasoning.

Rhet-Terms:*HDEBUG* If non-nil, doesn't complain about some normally illegal things, like destroying the root context. This flag is used generally to indicate that Rhet is in 'debug-mode', and thus it is exported. There are certain places the code will bind it to T to do useful things (like destroy the root context on a Assert:Reset-Rhetorical call!)

Rhet-Terms:New-Context *Name Parent-Name*
Creates a context with name Name and parent Parent-Name. All names accessible in the parent are also accessible in Name, unless explicitly removed (see **Remove-fact**). The function always returns T (errors are handled by exception).

Rhet-Terms:Pop-Context *Name*
Destroys the context named Name. An error is signaled if it is the parent of some other context. The function always returns T.

[1] An implementation decision that may change in the future: packages had a better UI than hashtables, and were thus thought easier to debug things in.

Rhet-Terms:Contextp *Name*

This will return Nil if Name is not a context (as defined by the user), and the Rhet-Terms:Context-Type structure and context if it is.

Rhet-Terms:Context-p *Context*

This will return Nil if Context is not a context, and the name of the context if it is (inverse to Contextp).

Rhet-Terms:Accessible-Context-P *Context1 Context2*

Returns T if Context1 is accessible from Context2, that is, is equal or a parent of context2.

Rhet-Terms:Destroy-Context *Name*

Like Rhet-Terms:Pop-Context, but the named context need not be a leaf. All children are also destroyed. Returns a list of the names of all contexts destroyed.

Rhet-Terms:Contexts &Optional *Root-Package*

Returns a list of the names of all known contexts, in tree form (*i.e.* specifying the inheritance hierarchy). The optional Root-Package should only be used internally by the function (for recursive calls).

Rhet-Terms:Symbol-Context *Symbol*

Returns the context a particular term is in.

Rhet-Terms:Context-Parent *Context*

Gets the Context's parent.

Rhet-Terms:Context-Children *Context*

Gets the Context's children.

Rhet-Terms:Convert-Name-To-Context *Name*

Convert the Name of a context to the context it represents.

Rhet-Terms:Convert-Context-To-Name *Context*

Convert the Context passed to the external name it is.

Rhet-Terms:Find-Term-In-Context *Term Context*

This function is abstractly like Find-Symbol, but we simulate Context's inheritance. That way, we can encode the rules exactly! So look up Term in the Context (package) and then recursively in each parent. Stop with the first one found.

Rhet-Terms:Atomic-P *Term1*

Returns non-nil only if Term1 has no arguments.

Rhet-Terms:Remove-Fact *Fact* &Key *Context*

Remove-fact removes the passed fact from the current (or specified) Context. Since it is illegal for particular fact to have more than one type, any inherent type specification is ignored. The fact is appropriately removed from any structures that point to it. If

Fact is not interned in the current (or specified) context, it is made to appear unbound in the current context, but left in it's normal context. This may involve adding the fact to the current context with a truth value of :Unbound. The function returns T if successful, Nil if the fact was not found. Note: It is illegal to even try to Remove-fact a fact that has a canonical name - equalities can only be removed by popping the contexts involved.

Rhet-Terms:Generate-Canonical-Name *EQ-Term Context* &Key *Localp*

This returns the term's canonical-name in the current (or specified) Context. Should none currently exist, it will create one (in which case the implicit type of the term will be used). The term need not be otherwise interned (*i.e.* by Add Term), as Generate-canonical-name will happily do so automatically. Setting the Local argument forces Generate-Canonical-Name to return the Term's canonical name local to this context, rather than by possible inheritance.

Rhet-Terms:HN-Union *Canonical-Name EQ-Term* &Key *Context*

HN-Union puts the term referenced by the EQ-Term into the class named by the Canonical-Name in the current (or specified) Context. The termneed not have been asserted by Rhet-Terms:Add-Term, as this function will happily do so automatically. HN-Union always succeeds[2], and so type and consistency checking should be done by the caller. It returns the canonical-name for the class everything was put into.

Rhet-Terms:HN-Find *Canonical-Name* &Key *Context*

Returns all terms in the class named that are accessible to the current (or specified) Context.

Rhet-Terms:Find-Closest-Children-Can-Union *FN-Term1 FN-Term2 Context*

This function takes two function terms and a Context and returns an alist of all child contexts and canonical names in those contexts, if any, one or the other has. The union part is that it will only report the canonical name of ONE of the two, which is what we need to process unions. (we will have to do a recursive union in all contexts that either of these terms are part of a class in.) It will ignore child contexts of other reported children. *i.e.* it returns an alist, where an entry is of the form context.canonical-name An example might be for the simple context inheritance $ROOT- > A- > B- > C$ where term FOO1 has a canonical name in each context, this function evaluated from context A would return ((B.can-name)) rather than ((B.can-name)(C.can-name)) since C is a child of B and so describing B suffices.

Rhet-Terms:Generate-Term-Index *Term*

Returns a Hash-Index for the Term. The Hash-Index will be how the Term will typically be looked up in the Context.

Rhet-Terms:Generate-Form-Index *The-Form*

Returns a Hash-Index suitable for finding all Terms that potentially match the Form.

[2] If it returns, it may use the condition system to signal an exception

Unexported constants and functions.

Rhet-Terms:Convert-To-Real-Name *Name*
Convert the outside world's view of a context's name to the internal name. The inverse of this function is Rhet-Terms:Convert-To-Outside-Name.

Rhet-Terms:Convert-To-Outside-Name *Name*
The inverse of Rhet-Terms:Convert-To-Real-Name.

Rhet-Terms:Context-Cleanup *Context*
This function cleans up references to Context in parent contexts. It is called by Rhet-Terms:Pop-Context-Internal. It is where we clean up child context references in the canonical name field of a Term, for instance, and eventually clean up TMS justifications that refer to children, *etc.*

Rhet-Terms:Blast-Symbol-In-Context *Term Context Root-Context*
This function follows the inheritance tree of a context up to the root (as passed) and interns Term in each (with the possible exception of the Root-Context, which will have it already unless it is Rhet-Terms:*Root-Context*.) It plays the game of setting the value of a particular instance of the Symbol to be the list (NIL PARENT'S-INSTANCE), so updating the parent works the way we want it to. It returns what to set the current symbol to. This function should not be called with Context of Rhet-Terms:*Root-Context* since that special case should be handled more simply. (used to determine recursion is ended).

Rhet-Terms:Install-Fact *Fact Key-Accessor-Function Context*
This function takes the fact, or function term we want to install in a context, and a function to apply to the CLTEVAL'd fact to get the data we want to invert on. (the key).

Rhet-Terms:Generate-Reference *FN-Term Referand* &Key *Context*
This function takes a FN-Term, sees if it has a canonical name, and if not generates one. It then creates a reference to the passed Referand for the canonical name. (the Referand is either a fact or a function term). It returns a list of contexts in which the FN-Term has canonical names.

Rhet-Terms:Delete-Reference *FN-Term Referand* &Key *Context*
This function is the opposite of Rhet-Terms:Generate-Reference, in that it takes a FN-Term and a Referand and undoes any reference to the Referand by the term's canonical class.

Rhet-Terms:Union-References *EQ-Term Can-Name Context*
This function takes a EQ-Term and a Can-Name which is local to the passed Context. The EQ-Term was referenced by a canonical name which is being unioned into the canonical destination name passed. In fact, the EQ-Term can be a canonical name, rather than some individual term.

Rhet-Terms:Update-Term-With-Can *EQ-Term Can-Name Context*

This function takes an equality Term and a Canonical Name which is local to the passed Context. The EQ-Term was in the set of a canonical name being unioned into the canonical name passed. Therefore, all we need to do is change the canonical name for the current context of the passed term to be the destination Can-Name. This adds a new entry for the current Context. Actually, that's not all we need to do - we must update the hashtable for the term as it is indexed by canonical names on the arglist. Note that we leave the hashtable entry for the plain arglist alone. This is only important if we are currently in the context the entry is made in, otherwise Rhet-Terms:Find-FN-Term will do the right thing.

Rhet-Terms:Update-Hashtable-With-Can *EQ-Term Context*

Called when any argument of the equality term has changed canonical classes. We need to reevaluate the arglist for updated canonical names in the passed context, and update the term's hashtable in this context for them. This may imply that this is the first hashtable for the head in this context, so we may have to do some linking...

13.3 How It Does It

First, contexts are implemented on the package system. The reason for this is twofold: packages (at least on the Symbolics™ and Explorer™s) are very efficiently implemented as hashtables. The advantage to using packages over hashtables themselves is debugging: it's a lot easier to be able to type the internal name of the context as a package name and the index and see what's there, rather than having to always look things up more or less manually in a hashtable.

The context related functions take a context name from the user and prepend "Hier-" to it it come up with the package name, and creates a package which has no :USE list. This is done to keep the system from complaining about multiple definitions, and making us shadow things explicitly, *etc.* (and it will still complain anyway, even when things are unambiguous, just to make sure you are aware there are two symbols with the same name!) So, the function Find-Symbol-In-Context does the inheritance for us. Using the WHO-AM-I constant, it can recursively look for a symbol in each parent if it doesn't find it in the passed context.

The "T" context is considered the most general one. Thus it is handled a bit specially by the functions: it's package is made the value of Rhet-Terms:*Root-Context* which is often checked to see if we are done looking for something, and there is no further to look.

The Rhet-Terms:Canonical-Name slot in a Rhet-Terms:FN-Term structure contains an alist whose keys are contexts, and whose values are canonical-names. This is to allow for a term to have a canonical name relative to the context.

The Type of an object can also vary with the context: consider the contexts above, where in the root context we know that Tweety is a bird and that penguins cannot fly, and that a complete partition of bird is penguins and non-penguins. If in FOO we have another object

Fred the penguin and we discover that Fred and Tweety are the same (so we make them equal) the type of Tweety is compatibly restricted to penguin. But Tweety's type in the root context cannot change, since the equality isn't present there (in fact Fred may not exist in the root — he is only in the FOO context — perhaps he is only a hypothetical entity). What we do is if a term has a canonical name we get it's type from the Rhet-Terms:Canonical structure, which varies by context. If it doesn't we use the type declared in the Rhet-Terms:FN-Term structure instead.

Canonical names are in some sense the most complex abstract item implemented by the Rhet-Terms package. The main reason is that they must be kept straight depending on context, and before and after unions are done. A particular term may have canonical names in several contexts as mentioned above, which is useful to determine when additional work must be done in a child context. That is, given that some function term, say [F A] is interned in the root context, and it is unioned with term [F C] in subcontext FOO1. If in the root we want to union [F A] with [G] we have to 'remember' somehow to update the union in the child context as well. It is not enough to make these things inherit: consider [A] and [B] in the root and [C] and [D] in the child with [EQ A C] and [EQ B D] (again in the child). Unioning A and B in the root would give us problems if we used inheritance – looking up the canonical name of B might give us B and A and C but require extra work to find D. So instead we just use these canonical names stored in the term in the most root-ward copy of the term (which we get via the functions Rhet-Terms:Find-Closest-Children-Can-Union and Rhet-Terms:Find-All-Children-Can) to tell us which child contexts will also require a union. We do the union in the child before we finish the one in the parent, because we don't want to give the child conflicting information via inheritance were we to do it in the parent first. The Rhet-Terms:Union-Recursive-Class function is used to do this work.

At that point we decide which of the two canonical classes we are going to union together we will copy to the new canonical name and which we will process in. (Copying is easy, processing involves updating references). We choose the class with the most references to be copied. We update the term structures with the new canonical names using Rhet-Terms:Update-Term-With-Can and call Rhet-Terms:Union-References to do the dirty work with the references.

Getting the canonical name from a function term basically involves either finding the canonical name associated with the current context, or calling Rhet-Terms:Get-Canonical recursively on the parent context until we do find it in the alist (or fail). Facts, themselves, have no canonical names.

The dirty work of Rhet-Terms:Union-References requires us to combine the reference fields of the two canonical classes HN-Union was called on. One set of references was simply moved to a new canonical name (the destination), the other must be processed in. Therefore, first check to see if there is an equivalent in the destination context's references for each referenced item, given that the canonical name contains all equivalences of arguments necessary. That is, if we had [F A] and [F B] and now are unioning A and B, (A $\Sigma\Sigma$ B's class) the passed FN-Term will be for [F A], the passe' Can-Name will be for B's class including A but only referencing [F B] plus anything else that was originally there. We want to detect that [F A] and [F B] are now equivalent, and do the recursive union. If

they are not, the reference can just be added as a reference for Can-Name. This involves checking the canonical names of the arguments to function terms with matching heads - we use Rhet-Terms:Can-Atom-Equivalent-P to make sure the heads are equal, and just check that the canonical names of each argument in this context are Eq to each other.

The basics of adding facts in the first place is relatively straightforward: The fact is interned in the appropriate package (context) and the head is also separately interned. A property on the head :Hash-All-Args is created and set to a hashtable whose keys will be the arglists of facts with this head and whose terms will the the facts. The fact is also interned by its index. Each of these (including the head) have as their value a list of all terms with this head, index, or marker. The last cdr of the list in this context, so to speak, is a pointer to the list in the parent context. In this manner, we can add facts with the same head to a parent context, and still pick it up from the head in the child - it will appear to be on its list! The hashtables are not so updated. Normally finding a fact involves recursive lookups in the hashtable for each head in successive parents until a term is found. The term is then returned unless it's truth value is :Unbound in which case the fact is considered to have been deleted and treated as if not found. (The :Unbound value is created by Rhet-Terms:Delete-Fact when the fact was added to some context, and deleted from a child - thus we still want to be able to access it from the parents of this child. We reintern the fact in the child context and set it's truth value to :Unbound which will hide it from this child and all of its children.)

The only other important thing to know is that each fact that is present on an arglist is given a canonical name, and the fact itself made a reference of this argument (via Rhet-Terms:Generate-Reference). Since often there will be more than one argument equivalent to the one when the fact was added (at least eventually) every time we union something which has a reference, we update the hashtable (as attached to :Hash-All-Args) and also use the canonical names of the arguments. This can get us into trouble, if we, say, union two facts such that their hashtable function would collide (*i.e.* on the same arglist we could get more than one fact). In this case rather than a fact, we put the canonical name into the value of the hashtable.

13.4 Still To Do...

- While we don't allow user specified hashing for purposes of unification (*i.e.* having the user specify which argument term of a function term will likely be matched on so hash on it, others being constant), right now we don't do much of anything automatically either. This makes things slower than they need to be, but upgrading it isn't a high priority either. *Functionality before Performance!*

- Too many functions take a fact and look up the canonical name, or take a canonical name, and have to double check that it's the right one for the context. Hashtables usually store the fact as the object of the hash, since storing the cname forces an entry into each context. Sometimes we get an entry in each context anyway, which isn't strictly needed. This all needs to be cleaned up for greater efficiency. The trade off is

putting the cname in the hash, and have more entries, and possibly losing information when attempting to do unifications. The actual pattern may matter, since `[P A]` may be asserted to be `Eq` to `[F T]` and thus should unify, (will be in the same canonical class) but that's not an excuse for `[F ?x]` to unify with `[P A]`: we have potentially lost information if `F` and `A` are distinct. That is, they should unify, but with different internal state as it were: `?x` is bound to `T`.

Appendix A

Installation of Rhet version 17.9

Installation is typical for systems distributed for Symbolics or Explorer hosts, *e.g.* there is a Defsystem file, and appropriate system and translation files should be installed in the site directory. Non-Lispm hosts should see the Porting Rhet appendix.

There is, however, one installation option, which determines how Rhet installs it's readtable. The parameter UI:*Make-Rhet-Readtable-Global* determines whether the readtable on the machine is replaced, or if only the Rhet process itself will use the Rhet readtable. If this term is not Nil, the symbol *Readtable* is Setf'd to the Rhet readtable, while if this value is Nil, the Rhet process (and the editor mode) merely binds it. Note that if this latter option is chosen, care must be taken with attempting to read rhet files in from a Lisp listener, or programmatic interface with Rhet, since these processes will need to first bind their *Readtable*s as well. Last, it is a function of the User:*Rhet-Initializations* list, which is normally run immediately after loading Rhet, to set up this readtable. See section B.2, for more details on initialization lists.

A.1 Special Instructions based on machine type

A.1.1 Symbolics

Rhet should load and run on a Symbolics "out of the box" if your distribution was via a "Distribution Tape". TeX (available from SLUG) is also required to use the distributed defsystem. If you do not choose to get TeX, you may delete references to the manual from the defsystem, and than Rhet should be able to load. (The machine really only needs to know about TeX file types, rather than have the entire TeX distribution).

A.1.2 Explorer

Rhet has not been locally supported on the explorer since version 16, however, other that the user interface code (windowing), it should be easily portable.

Note that you must load the G7C system (provided in the public directory in releases beginning with 4.1) in order to load Rhet.

Certain other public domain subsystems may be loaded by the Rhet defsystem file; these will be found in the utilities subdirectory.

Appendix B

Porting Rhet

This section is vastly underspecified. As implementations of CL are extended with standardized error handling, objects, etc. divergences from the standard should become less necessary, and as compiler technology improves, even less so (efficiency hacks will no longer be needed). Version 17 includes usage of CLOS, we expect version 18 will be released both on Symbolics systems, and under Allegro Common Lisp. Note that version 16 was the last version to be tested and run on an Explorer system. While explorer-specific code has not purposely been stripped, it will be not be maintained in the future and may break during the port to Allegro.

B.1 Overall comments

Note that the source code is occasionally multi-fonted (typically using the Symbolics format). This may cause problems on non-lispms (or even non-Symbolics machines). Font information will appear to ASCII devices to be a 006 (binary) character[1], followed by other font-specific information. For a good example of this, see the copyright.text file in the home directory of the distribution. On a Symbolics, the copyright line appears as:

```
;;; © Copyright (C) 1990, 1989, 1988, 1987, 1986 by the University of
Rochester.  All rights reserved.
```

For the most part, font information appears inside of comments and should not effect compilation.

B.2 Initialization Lists

Rhet takes advantage of the Lispm initializations lists to defer certain processing until after all of Rhet is loaded. Such forms are put on the User:*Rhet-Initializations* list and are expected to be executed only once, immediately after loading Rhet. Other forms may also

[1] In ASCII, this is control-F or "ack".

be put on the system's cold or warm boot list. The cold boot list is expected to be executed after a cold boot, while the warm boot list is executed after every warm boot and all cold boots. Forms are added to these initialization lists via the SI:Add-Initialization function, and run using the SI:Initializations function.

B.3 Package Handling Functions

Rhet makes use of some package handling functions available on the lisp machines, that are possibly implementation dependant (they are not in basic common-lisp). Specifically:

Pkg-Kill This deletes a package from the system and uninterns any symbols that had that package as their home package.

Mapatoms A mapping form that maps a function over all of the symbols in a package.

If by the time you read this you are running version 18, note that some substitute for these functions will be provided, or the calls eliminated.

B.4 High Level User Interface

Virtually everything in the window-interface directories are extremely implementation dependant, as they depend on the host machine's model of processes, windows, the mouse or other i/o interface, prompting for typed expressions, documentation and assistance presentation, *etc.*. Rhet can be loaded and run without these functions, but a complete implementation should have equivalent functionality running on the host machine to facilitate rapid prototyping and debugging of systems implemented on Rhet.

The Rhetorical system window interface is an interactive semi menu-driven program that allows the user to interactively use the Rhet System. It consists of a frame with several configurations (currently) and a process, associated with the frame. The frame is selectable via <select>-r.

For the most part, this code is so machine dependent it is not further documented here. You need to be a systems hack to make changes to this for a non lispmachine, since it is based on the presentation model of the machine. The symbolics version makes heavy usage of presentation types, and uses the DW:Define-Program-Framework style of defining windows and commands. The explorer doesn't have a command processor, so instead it uses constraint frames and menus. We expect future versions of Rhet (starting with version 18) to use CLIM (Common Lisp Interface Manager), and any environement that can run it will be able to run Rhet's generic interface.

Bibliography

[Abelson and Sussman, 1985] Harold Abelson and Gerald Jay Sussman, *Structure and interpretation of Computer Programs*, MIT Press, Cambridge, Massachusetts, 1985.

[Allen and Miller, 1986] James F. Allen and Bradford W. Miller, "The HORNE Reasoning System in COMMON LISP," Technical Report 126, University of Rochester, Computer Science Department, August 1986, Revised.

[Allen and Miller, 1989] James F. Allen and Bradford W. Miller, "The Rhetorical Knowledge Representation System: A User's Manual," Technical Report 238 (rerevised), University of Rochester, Computer Science Department, March 1989.

[Bruynooghe and Pereira, 1984] M. Bruynooghe and L. M. Pereira, "Deduction Revision by Intelligent Backtracking," In J. A. Campbell, editor, *Implementations of Prolog*. Ellis Horwood Limited, 1984.

[Bruynooghe, 1982] Maurice Bruynooghe, "The Memory Management of PROLOG Implementations," In K. L. Clark and S.A. Tarnlund, editors, *Logic Programming*. Academic Press, 1982.

[Campbell, 1984] J. A. Campbell, editor, *Implementations of Prolog*, Ellis Horwood Limited, West Sussex, England, 1984.

[Clark and Tarnlund, 1982] K.L. Clark and S.A. Tarnlund, editors, *Logic Programming*, Academic Press, New York, 1982.

[Cox, 1984] P. T. Cox, "Finding Backtrack Points for Intelligent Backtracking," In J. A. Campbell, editor, *Implementations of Prolog*. Ellis Horwood Limited, 1984.

[Fagin, 1984] Barry Fagin, "Issues in Caching Prolog Goals," Technical Report UCB/CSD 84/204, University of California at Berkeley, Computer Science Division, November 1984.

[Haridi and Sahlin, 1984] S. Haridi and D. Sahlin, "Efficient implementation of unification of cyclic structures," In J. A. Campbell, editor, *Implementations of Prolog*. Ellis Horwood Limited, 1984.

[Hopcroft and D., 1979] John E. Hopcroft and Ullman Jeffrey D., *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[Kahn and Carlsson, 1984] K. M. Kahn and M. Carlsson, "How to Implement PROLOG on a Lisp Machine," In J. A. Campbell, editor, *Implementations of Prolog*. Ellis Horwood Limited, 1984.

[Keene, 1989] Sonya E. Keene, *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, Addison-Wesley, Reading, Massachusetts, 1989, ISBN 0-201-17589-4.

[Koomen, 1989] Johannes A.G.M. Koomen, *Reasoning About Recurrence*, PhD thesis, University of Rochester, July 1989, Also TR 307.

[Kornfeld, 1983] W. A. Kornfeld, "Equality for Prolog," In *Proceedings, 8th IJCAI*, Karlsruhe, W. Germany, August 1983.

[Mellish, 1982] C. S. Mellish, "An Alternative to Structure Sharing in the Implementation of a PROLOG Interpreter," In K. L. Clark and S.A. Tarnlund, editors, *Logic Programming*. Academic Press, 1982.

[Miller, 1989] Bradford W. Miller, "Rhet Programmer's Guide," Technical Report 239 (rerevised), University of Rochester, Computer Science Department, March 1989.

[Miller, 1990] Bradford W. Miller, "The Rhetorical Knowledge Representation System Reference Manual," Technical Report 326, University of Rochester, Computer Science Department, November 1990.

[Pearl, 1984] Judea Pearl, *Heuristics — Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, Reading, Massachusetts, 1984.

[Rees and Clinger, 1986] Jonathan Rees and William et. al. Clinger, "Revised^3 Report on the Algorithmic Language Scheme," *SIGPLAN Notices*, 21(12), December 1986.

[Steele Jr., 1990] Guy L. Steele Jr., *Common Lisp the Language 2/e*, Digital Press, 1990, ISBN 1-55558-042-1.

[Sterling and Shapiro, 1986] Leon Sterling and Ehud Shapiro, *The Art of Prolog: Advanced Programming Techniques*, MIT Press Series in Logic Programming. MIT Press, Cambridge, Massachusetts, 1986.

[van Caneghem and Warren, 1986] Michal van Caneghem and David H. D. Warren, editors, *Logic Programming and its Applications*, Ablex Publishing Corporation, Norwood, New Jersey, 1986.

[Wada, 1985] Eiiti Wada, editor, *Logic Programming '85*, New York, July 1985, Springer-Verlag, Proceedings of the 4th Conference: Tokyo, Japan.

[Warren, 1977a] David H. D. Warren, "Implementing PROLOG — Compiling Predicate Logic Programs: Volume 1," Technical Report 39, Department of Artificial Intelligence, University of Edinburgh, May 1977.

[Warren, 1977b] David H. D. Warren, "Implementing PROLOG — Compiling Predicate Logic Programs: Volume 2," Technical Report 40, Department of Artificial Intelligence, University of Edinburgh, May 1977.

[Warren, 1986] David H. D. Warren, "Optimizing Tail Recursion in Prolog," In Michal van Caneghem and David H. D. Warren, editors, *Logic Programming and its Applications.* Ablex Publishing Corporation, Norwood, New Jersey, 1986.

Index